COURANT COMPUTER SCIENCE PUBLICATIONS

COURANT COMPUTER SCIENCE NOTES

A101 ABRAHAMS, P. *The PL/I Programming Language*, 1979, 151 p.

C66 COCKE, J. & SCHWARTZ, J. *Programming Languages & Their Compilers*

D86 DAVIS, M. *Computability*, 1974, 248 p.

M72 MANACHER, G. *ESPL: A Low-Level Language in the Style of Algol*, 1971, 496 p

M81 MULLISH, H. & GOLDSTEIN, M. *A SETLB Primer*, 1973, 201 p.

S91 SCHWARTZ, J. *On Programming: An Interim Report on the SETL Project. Generalities; The SETL Language & Examples of Its Use*, 1975, 675 p.

S99 SHAW, P. *GIVE--A Programming Language for Protection and Control in a Concurrent Processing Environment*, 1978, 668 p.

S100 SHAW, P. ", Vol. 2, 1979, 600 p.

W78 WHITEHEAD, E.G. Jr. *Combinatorial Algorithms*, 1973, 104 p.

COURANT COMPUTER SCIENCE REPORTS

1 WARREN, H. Jr. *ASL: A Proposed Variant of SETL*, 1973, 326 p.

2 HOBBS, J. R. *A Metalanguage for Expressing Grammatical Restrictions in Nodal Spans Parsing of Natural Language*, 1974, 266 p.

3 TENENBAUM, A. *Type Determination for Very High Level Languages*, 1974, 171 p

4 OWENS, P. *A Comprehensive Survey of Parsing Algorithms for Programming Languages*, 662 p. +. ..

5 GEWIRTZ, W. *Investigations in the Theory of Descriptive Complexity*, 1974, 60 p.

6 MARKSTEIN, P. *Operating System Specification Using Very High Level Diction*, 1975, 152 p.

7 GRISHMAN, R. (ed.) *Directions in Artificial Intelligence: Natural Language Processing*, 1975, 107 p.

8 GRISHMAN, R. *A Survey of Syntactic Analysis Procedures for Natural Language*, 1975, 94 p.

9 WEIMAN, CARL *Scene Analysis: A Survey*, 1975, 62 p.

10 RUBIN, N. *A Hierarchical Technique for Mechanical Theorem Proving and Its Application to Programming Language Design*, 1975, 172 p.

11 HOBBS, J.R. & ROSENSCHEIN, S.J. *Making Computational Sense of Montague's Intensional Logic*, 1977, 41 p.

12 DAVIS, M. & SCHWARTZ, J. *Correct-Program Technology/Extensibility of Verifiers*, with *Appendix* by E. Deak, 1977, 146 p.

13 SEMENIUK, C. *Groups with Solvable Word Problems*, 1979, 77 p.

14 FABRI, T. *Automatic Storage Optimization*, 1979, 91 p. ..

NOTES: Available from Department LN. Prices on request.
REPORTS: Available from Ms. Lenora Greene.
Free, except #1,3,4,6,7,8,10 .. out of print: xeroxed at going rate.

COURANT INSTITUTE OF MATHEMATICAL SCIENCES
251 Mercer Street
New York, N. Y. 10012

COURANT INSTITUTE OF MATHEMATICAL SCIENCES

Computer Science NSO-14

AUTOMATIC STORAGE OPTIMIZATION

Janet Fabri

1979

Report No. NSO-14 prepared under
Contract Number NSF-MCS 76-00116 with
the National Science Foundation

CONTENTS

* Courant Institute Library has Appendices.

PREFACE

I would like to express my deep appreciation to Professor Robert Dewar for his supportive counsel and wise guidance during this dissertation effort, and for patiently reading through several versions of this document. I would also like to thank Professor Jacob Schwartz for his sage advice. Most stimulating to this work were a number of (separate) technical discussions that were held with Professor Martin Golumbic, Dr. Alan Hoffman, Gregory Chaitin and John Cocke, for which I am very grateful. Furthermore, this work would not have been at all possible without the cooperation and practical support provided to me by my management at IBM; in particular, I am indebted to my immediate manager, Dr. Robert Wilkov, for this support and for the constructive suggestions he provided to expedite my efforts. Finally, I thank the people close to me for their patience and support.

1. INTRODUCTION

1.1 Motivation

Program optimization theory is the study of techniques for improving the execution characteristics of automatically compiled programs. As others (CS70, AhHU77) have observed, the term optimization is misleading; program improvement is more apt. There is, after all, no such thing as an absolutely optimum program. In one environment it is object execution time that is critical; in another, object program size; in a third, compiler execution time. Different hardware architectures favor different optimizations. Even operating system considerations, such as input/output design and storage management design, can affect optimization trade-offs.

Most studies of optimization techniques for higher level languages have focused on improving execution time of generated programs, often at the expense of increased storage. When storage optimization has been addressed, it is usually in conjunction with time optimization, such as in instruction-reducing code transformations. In Section 1.3, two existing storage-optimizing compilers are discussed.

The rising popularity of minicomputers and microprocessors suggests that the time has come to take a closer look at the problem of automatic storage optimization. Because lack of space has always been a problem in the small machine environment, the proliferation of small machines implies the increasing importance of the problem. Although the decreasing cost of memory may mitigate this trend, a variant of Murphy's law ensures that program size will always increase faster than the available storage; in short, programmers will always write programs that don't fit.

Even without the advent of the small computer, such an investigation would be warranted from a language point of view. Today, almost all programming languages include the notion

of storage in the concept of a variable, and most compilers maintain a one-to-one mapping between variables and storage. This means that, in a tight storage situation, the programmer must overlap storage by deliberately using a single variable for more than one purpose, to the detriment of the clarity and reliability of the program. A desirable goal would be a language in which variables had no storage connotation, but where the processor performed all storage allocation decisions, guaranteeing only the integrity of the variable. This is an important goal, since, unlike the case of time optimization, where the scope of a coding trick is relatively local, a storage-optimizing coding trick often obscures the entire program.

Storage optimization is also applicable in virtual storage systems where a decrease in program size may result in a smaller page requirement for the object program, with consequent improvement in program execution time. The related question of organizing procedures of a given size so as to minimize interpage transitions has been treated by Kernighan (Ker70, Ker71).

Another related question arises when a computer with multiple memory classes is considered. Each class has a maximum size and a unit access cost. Without even considering storage minimization, the choice of storage area for each variable will affect the program execution time. This problem has been elegantly solved by Warren (Wa78).

1.2 Optimization Techniques and Compiler Structure

Compiler optimization techniques, together with the rest of compiler technology, have evolved from a collection of largely ad hoc techniques into a body of systematic theory and practice.

The FORTRAN H compiler (LowM69) was one of the earliest systematic implementations of an optimizing compiler. Introducing the notion of back-dominance, the compiler was able to perform redundant expression elimination and code motion, among other optimizing transformations.

The invention of the interval concept by John Cocke (C71) laid the groundwork for most of the ensuring research in systematic program optimization. Using the graph-theoretic notion of an interval, Cocke and Allen demonstrated a procedure (Al71,Al74,AlC72b,AlC76) that analyzes the data flow relationships in a program and generates information useful for systematic code transformations such as redundant expression elimination, constant progagation, and code motion (Ken1,C70).

Kildall (Ki73) introduced a theoretic model for data flow analysis problems using lattice algebra. In most cases, a simple, general iterative algorithm can be shown to converge. Kam, Ullman, Hecht and others (KaU76,HeU75) have extended these results. In addition, node listing techniques have been used (e.g., Ken75). Fast and simple algorithms for data flow analysis are now known (GrW76,U73).

Today, program optimization is heavily investigated in a number of directions, such as interprocedure optimization (B77), graph grammars (KenZ), symbolic evaluation (Rle77), data type determination (Te74), reduction in strength (PSchw77), and data structure choice (Schw75), among many others. Reference (He77) is a good textbook on the subject of data flow analysis and its uses. Other books that include information on program optimization are (AhHu77,CS70,Scha73,WJWHG75).

The standard optimizing compiler performs a syntactic and semantic analysis of the program, producing an internal form on which optimization can take place. This internal form includes one or more program flow graphs, together with node-specific information on the use and definition of the program's data. A data flow analysis phase determines, via interval analysis or an iterative method, information such as the following: (1) the set of expressions available at each node; (2) the set of variables live at each node; (3) definition-use chains; (4) use-definition chains. Using this information, code-improving transformations (AlC72a), such as redundant expression elimination and constant propagation, are applied in subsequent phases. Machine-dependent optimization may be performed during the final code generation phase.

1.3 Towards a Storage-Optimizing Compiler

Live information is of the greatest importance for storage optimization. If a variable is not live in a portion of the program, its storage can be used by another variable that is live there. Live value analysis is thus an indispensable function of a storage-optimizing compiler.

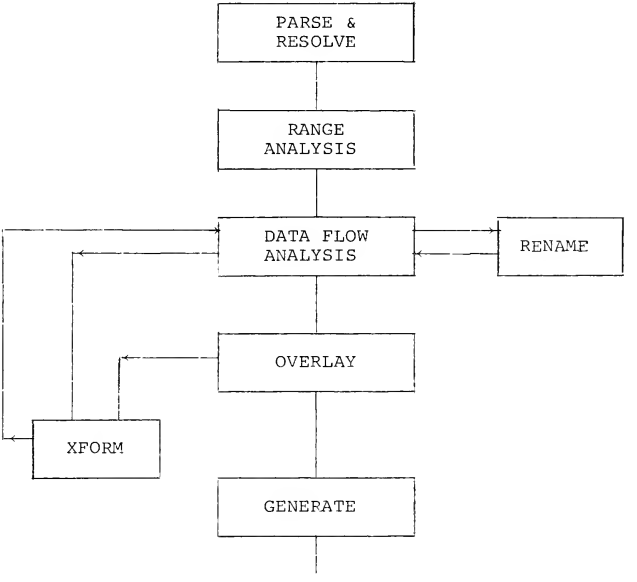
Additional data flow information is needed if the compiler performs storage-optimizing code transformations. As we will demonstrate in the next chapter, such program transformations can improve data overlay possibilities. Available expression information, definition-use and use-definition chains are needed.

Two existing compilers — Alpha and Bliss — have taken limited but quite different approaches to storage optimization. In the Alpha compiler (Y71), Yershov tackled the problem of automatic data overlay determination. From the information collected by a live value analysis, the Alpha compiler builds a conflict graph (see Section 1.5), and uses a combined coloring and packing heuristic to determine sets of overlaying variables. Yershov's overlay heuristic will be discussed in detail in Section 3.2. The Alpha compiler, however, did not address storage-optimizing code transformation.

In the Bliss compiler (WJWHG75), attention is focused on optimizing instruction storage and temporary storage, but overlay of aggregate variables is not addressed. Data flow analysis is performed in an early phase. In the next phase, three basic functions are performed in order to prepare for the production of shortened code sequences: (1) the "general shape" of the object code is determined; (2) the "cost" of each linear program segment is estimated; (3) code is reordered heuristically so that a minimal number of registers are required. The assignment of temporary variables to storage (or registers) is performed in three subsequent phases by ranking the temporaries via a figure of merit, and assigning them in ranked order to shared locations. (The Bliss interpreters did not use the conflict graph to simplify the representation of live variable conflicts.) A final phase

generates the code and performs peephole optimization.

In this work we are interested in a compiler that performs both automatic data overlay determination and storage-optimizing code transformation. For such a compiler, the following structure is proposed:



After preliminary processing, a range analysis procedure (Harr75) is used to determine variable subreferences and their disjointness and covering properties. The use of range analysis is discussed in Section 2.2. A generalized array data flow analysis procedure, similar to the one implemented in the current effort (see Section 7.4) is performed next. Automatic overlay procedures, including the ones implemented in the current effort, are discussed in detail in Chapter 3.

Renaming transformations improve overlay opportunities by inserting new variables into the program to assume some, but not all, of the critical live conflicts. A canonical renaming transformation is introduced in Chapter 4 of this dissertation. Other code transformations eliminate conflicts between a particular pair of variables either before or after overlay determination, as described in Chapter 5. Examples of these transformations are given in Chapter 2, and implementations of these algorithms have been included in the current effort.

1.4 An Empirical Investigation

A major deficiency in the development of optimizing compilers has been the lack of empirical data on the way programming languages are actually used in production programs. Optimizing transformations are often selected or discarded for a particular compiler based on the compiler designer's feelings about what programs are like and which transformations seem appealing.

A significant departure from this trend is Knuth's study (Kn71), which gathered information on the frequency of use of various FORTRAN facilities, as well as on the reducibility of programs. Unfortunately, this study postdates the development of several of the widely used optimizing compilers.

In order to gather information on storage optimization, an empirical investigation has been undertaken, using a tool built for this purpose called SOCRATES (the Storage Optimization Code Reorganization And Transformation Experimental System) to study the amenability of typical programs to various storage optimization techniques. SOCRATES has been implemented as part of this effort, and several moderate-sized programs have been analyzed. SOCRATES is described in Chapter 7, and the experimental results are given in Chapter 8.

1.5 Definition of Terms

In this work, we will be referring to a number of terms which are summarized below:

FG will denote the flow graph of a program P. FG is a pair (ND, ED), where:

ND is the set of nodes of the program. These may be single operations, single statements, basic blocks, procedure blocks, or whatever unit has been determined to be fundamental for the storage analysis.

ED is the set of edges (I,J) in the program flow graph, for I and J in ND.

A path is a sequence of edges <<E(i)>> such that the tail of edge E(i) is equal to the head of edge E(i + 1).

A depth-first post-order of ND will be referred to extensively in this paper. A depth-first search is an efficient search of a graph which produces a node ordering that has many useful properties (Ta72,Ta74,HeU75). We will exploit these in SOCRATES.

VARS will denote the set of variables in the program. For each variable V in VARS, |V| or SIZE(V) will denote the declared size of V. For any subset SV of VARS, the size of SV, |SV|, is the sum of the sizes of the elements of SV. Clearly, |VARS| is an upper bound on the program's data storage requirement.

A subreference S of an aggregate variable V is defined to be a specific set of scalar elements of V. REFS will denote the set of references in the program. Each reference R in REFS is a pair (V,S), where V is the variable and S is a subreference of V. By convention, if S is null, the reference is to the entire variable V.

If S1 and S2 are both nonnull, but the intersection of S1 and S2 is null, then (V,S1) and (V,S2) are said to be disjoint. If (V,S1) and (V,S2) are not disjoint, they are said to be conjoint. A set of subreferences S1,...,Sk is said to cover a reference R if every scalar element of R is in at least one subreference Si, i=1,...,k. Each Si is called a subreference of R. Every reference (V,S) is a subreference of V.

For each node I in ND, let USES(I) be the set of references used at node I.

For each node I in ND, let DEFS(I) be the set of references redefined at node I. If a reference is redefined at that node, every value in the specified subreference is altered.

For each node I in ND, let MODS(I) be the set of references modified (or redefined) at node I. If a reference is modified at that node, some, but not necessarily all, of the values in the specified subreference are altered.

A reference (V,S) is said to be live at a node I if there is a DEFS-clear path in FG from node I to any node J at which (V,S), or some conjoint reference (V,S'), is in USES(J). A variable V is said to be live at node I if any of its subreferences are live. The set of variables live at node I is denoted by LV(I).

A variable V is said to be active at node I if it is in LV(I), or if any of its subreferences are in DEFS(I), MODS(I), or USES(I). The set of variables active at node I is denoted by ACT(I). Intuitively speaking, ACT(I) is the set of variables which must have storage reserved at node I. LS denotes the set of active sets ACT(I), over I in ND.

A loop is a subset of nodes in FG with a single entry and a single exit, which contains a back edge from the last node (in depth-first post-order) to the first node, and may contain other back edges. A variable V is said to be active at loop L if it is active at some node of L.

The set of nodes or loops at which a variable IV is active is denoted by LNACT(IV). The set of nodes or loops at which a variable IV is used is denoted by LNUSES(IV). The set of nodes or loops at which a variable IV is modified (or redefined) is denoted by LNMODS(IV).

A modification of a reference R within a loop L is said to propagate backwards into a definiton of R if every element of R is defined before any use in L. Such a propagation will result in R being dead (i.e., not live) on entry to loop L.

A modification of a reference R within a loop L is said to propagate forwards into a definition of R if every element of R is defined in L. Such a propagation will result in R being available on exit from loop L.

We define MAXLIVE to be the maximum over the elements ACT(I) of LS of |ACT(I)|. Intuitively speaking, MAXLIVE is the maximum storage needed at any particular node. It is a lower bound for the program's overlay storage requirement.

The array conflict graph CG is defined: the nodes of CG are the elements of VARS. An edge (X,Y) is in CG if there is some node I in ND such that X is in ACT(I) and Y is in ACT(I). Each node has associated with it a weight, corresponding to the size of the variable. A clique is a subset of the graph in which every node is connected to every other node in the subset.

The conflict graph is central to automatic storage optimization. An edge connects a pair of nodes if and only if the variables are active simultaneously and may therefore not share storage. The size of any subgraph of the conflict graph is defined to be the sum of the weights of the nodes. If the cliques of the conflict graph are examined, a clique whose size is the largest is called a "maximum clique", and its size is called the "maximum clique size". Since, by definition, no pair of variables in the maximum clique may share storage, the maximum clique size is a lower bound on the storage overlay requirement.

More precisely, let CS denote the set of cliques in CG, and define MAXCLIQ as the maximum value of |C| over the elements C of CS. Note that MAXCLIQ may exceed MAXLIVE. For example, if X and Y are active simultaneously, Y and Z are active simultaneously, and X and Z are active simultaneously, but X, Y, Z are never active simultaneously, MAXLIVE < MAXCLIQ, and separate storage locations must be reserved for each variable. It is shown in Chapter 4 that the program can always be transformed so that MAXLIVE is reduced to MAXCLIQ.

In Chapter 3, the definitions of a coloring map and of the chromatic number CHR(CG) are generalized for array conflict graphs in such a way that a coloring of the conflict graph corresponds to a storage layout and CHR(CG) corresponds to the minimum-size storage layout.

1.6 Structure of the Paper

The aim of this paper is to delineate some fundamental issues in the desing of a storage optimizing compiler that performs automatic data overlay determination and storage-optimizing code transformation. In Chapter 2, highlights of the subject are provided by means of examples that illustrate means of achieving storage savings.

Automatic data overlay, the subject of Chapter 3, is the central problem of storage optimization; all subsequent techniques are aimed at improving the opportunities for automatic data overlay. This problem is stated formally, and an exact (but exponential) algorithm for its solution is presented. The overlay problem is considered in relation to several NP-complete problems. Past work on overlay heuristics is surveyed, and the family of heuristics that has been implemented in SOCRATES is presented. An upper bound for the extended chromatic number has been demonstrated by A. Hoffman, and the polynomial-time algorithm that realizes that bound is presented, together with a proof of the bound. This result is significant for storage optimization because program transformations can be aimed at lowering this upper bound as well as the lower bound MAXCLIQ.

Program transformations that reduce the conflict graph's maximum clique size usually improve the results of an overlay heuristic. Two transformations are presented in Chapter 4 that reduce the maximum clique size to the program's maximum active set size by introducing new variables into the program to assume some, but not all, of the critical live variable conflicts. These are generalized to a single, canonical transformation which transforms a program so that the maximum clique

size is always reduced to the maximum active set size. This result is significant because it provides a systematic basis for the automation of an ad hoc programming technique. In particular, this algorithm is important for register optimization, since it determines where "move-register" operations are to be profitably inserted to reduce the number of required registers. This transformation can also be used to break up other cliques in the conflict graph.

In Chapter 5, basic code-modifying transformations are considered. The code modification problem is formulated as a generalization of an NP-complete register optimization problem. A goal-directed SOCRATES procedure that determines what program nodes should be split, copied or moved in order to eliminate edges in the conflict graph is presented. If the conflict graph or the overlay results are used to select nodes, the potentially combinatorial size of the storage-minimizing code transformation problem is reduced to more manageable proportions. Redundancy equations that express safety constraints for redundant code insertion and code motion are also presented in Chapter 5, and profitability tests are described.

In Chapter 6, transformations and techniques that have not been eliminated in SOCRATES are described. Loop fusion, rank reduction, redundant code elimination and data spill are illustrated by example and data fragmentation, instruction block overlay and interprocedural overlay are discussed. The transformations of Chapters 5 and 6 are also relevant to register optimization.

In Chapter 7, the SOCRATES effort is described, and the Storage Optimization Language (SOL) specified. In SOCRATES, the redundancy equations for live value analysis and available value analysis have been generalized for arrays and array subreferences, and this is included with the program analysis phase description in Section 7.4. The SOCRATES implementation also includes versions of the overlay heuristics described in Chapter 3, the renaming transformations described in Chapter 4, and the code modification transformations described in Chapter 5

In Chapter 8, the empirical data gathered by SOCRATES to date is summarized. In Chapter 9 the conclusions of this study are reviewed, and directions for further study are indicated.

Throughout this dissertation, algorithms are specified in SETL/2 (De78). SOCRATES itself has been programmed in PL/1, and a listing of this program, together with sample program runs, are given in the Appendix.

2. HIGHLIGHTS OF STORAGE OPTIMIZATION

Before beginning our study of automatic storage optimization techniques, it is instructive to consider the manual techniques used by the programmer to achieve storage economy. From such a catalog, it will be possible to identify those techniques suitable for automation.

Many languages have a facility like the FORTRAN EQUIVALENCE statement that permits the programmer to specify the overlay of static or program data. In languages where this facility is limited or nonexistent, the programmer can overlay data by reusing variable names. In either case, the resulting program is difficult to read, difficult to modify and often unreliable. These techniques are also applicable to automatic variables local to a given block (e.g., in ALGOL or PL/1). Data overlay is discussed with example in Section 2.1, and the relevance of range analysis to data overlay is discussed in Section 2.2.

Renaming techniques can be used to modify the program so that more overlay opportunities materialize. Examples of these transformations are given in Section 2.3.

In (AlC72a), Cocke and Allen present a catalog of optimizing transformations. Some of these transformations and their generalizations can be used to enhance the opportunities for data overlay. Examples are given in Section 2.4.

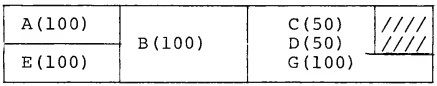
Other techniques, which are usually more costly in execution time, are discussed in Chapter 6.

2.1 Data Overlay

The following PL/1 program segment illustrates an opportunity for data overlay.

```
DCL A(50,2),B(50,2),C(50),D(50),E(50,2),G(50,2);
GET LIST(B,C);
A = B / C;
D = A(*,1)**2 + B(*,2)**2;
E = B(*,1)**2 + D;
G = SQRT(E);
PUT LIST(E,G);
```

If the arrays declared above were to be arranged sequentially in storage, they would occupy 500 units. However, a storage-conscious programmer (using overlay defining), or a storage-conscious compiler, might try to overlay storage in some manner. For example, it might be observed that A and E are never around at the same time, and neither are C, D and G. Thus, the following overlay of storage is possible:



Total: 300

A storage-optimizing compiler can perform data overlay automatically, by using live value information. In the above program, A and E are not active simultaneously, so they can share storage. Similarly, C, D and G can all share storage.

Storage can be conserved even further by increasing the amount of overlaid storage:

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

Total: 250

Alternatively, since B and G are also not active simultaneously, they can share storage. More storage is saved by overlaying G on B than by overlaying G on D. In general, the problem may be difficult for a programmer to solve, since it involves examining a large number of overlap choices. As we shall see, the task is also difficult for a storage optimizer. In Chapter 3, the automatic overlay problem will be considered in detail.

2.2 The Relevance of Range Analysis

Range analysis techniques can be used to refine information on the live properties of variables. Consider the following example:

```
        DCL  A(20);
             .
             .
             .
(1)     DO I = 1 TO 20;
        A(I) = ...
        END;
             .
             .
             .
           DO WHILE(SW);
             .
             .
             .
(2)           DO J = 1 TO 20;
                   = ... A(J) ...
(3)           END;
             .
             .
             .
(4)              DO K = 1 TO 20 BY 2;
                 A(K) = ...
                 END;
                 DO L = 2 TO 20 BY 2;
                 A(L) = ...
                 END;
             .
             .
             .
(5)     END;
```

In the absence of range analysis, A would have to be assumed live, and have storage reserved for it, between lines 1 and 5. However, a range analysis procedure could discover that the loop on line 4 assigns to the odd-numbered elements of A and the succeeding loop assigns to the even-numbered elements of A, so that A is dead between lines 3 and 4 where it can share storage with a variable that is live only in that interval.

This example illustrates the desirability of a generalized live value analysis procedure that recognizes portions of an array (i.e., in this case, the odd-indexed elements, the even-indexed elements), and their interrelationships (the two portions are mutually disjoint and cover all of A).

In Section 1.5, a "subreference" was defined in order to express the notion of such a portion of an array.

Generalized available value analysis produces refined information for storage-optimizing code modification, and is also facilitated by range analysis.

It is noteworthy that the element-by-element modifications in the first, third and fourth loops result in a definition of the entire array in the first loop, and of each subreference in the other loops. This concept was captured in the notion of propagation (see Section 1.5).

2.3 Renaming Transformations

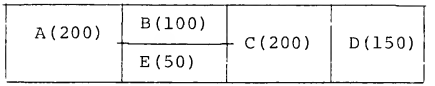
In this and succeeding sections, examples of code transformations will be presented, using a subset of the SOL language. This language, which is specified in Section 7.3, expresses the characteristics of a language that are essential for storage optimization. Each statement may be thought of as corresponding to a program statement, and the syntax is PL/1-like. For the purpose of these examples, the options appearing in the statement are: the USE option, describing the variables used in the statement; the SET option, describing the variables redefined in the statement; and the GOTO option, describing the successor statements, used only when the physically succeeding statement is not the only successor. The DEF statement describes the sizes of the variables in this subset.

2.3.1 Unsharing

Sometimes, the programmer's use of a single variable for several purposes interferes with storage overlay. Consider the following program:

```
        DEF A(200),B(100),C(200),D(150),E(50);
1:      SET(A,B);
2:      SET(C) USE(A,B);
3:      SET(D) USE(B,C);
4:      SET(A) USE(C,D);
5:      SET(E) USE(A,C);
6:      USE(A,E);
```

The best storage utilization for this program is the following:

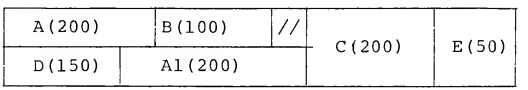


Total: 650

The only storage sharing possible is between B and E. However, 50 units of storage can be saved if variable A is "unshared".

```
    DEF A(200), A1(200), B(100), C(200), D(150), E(50);
1:     SET(A,B);
2:     SET(C) USE(A,B);
3:     SET(D) USE(B,C);
4:     SET(A1) USE(C,D);
5:     SET(E) USE(A1,C);
6:     USE(A1,E);
```

The following storage assignment is now possible:



Total: 600

A new variable, A1, can be used in statement 4, because A is dead on entry to statement 4. If A and A1 are assigned different storage locations, D can overlay A, resulting in the storage utilization pictured above. An unsharing transformation has been performed on the program, improving the opportunities for storage overlay.

2.3.2 Repositioning

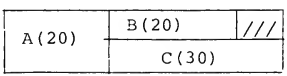
Repositioning is a technique that will be familiar to any assembly-language programmer who has ever tried to fit a program into tight storage. It involves moving data from one variable (or register) to displace another whose current value is not needed so as to make room for a third variable's value. Consider the following example:

```
        DEF A(20), B(20), C(30);
1:        SET(A,B);
2:        USE(A,B) GOTO(3,4);
3:           SET(C) USE(A) GOTO(5);
4:           SET(C) USE(B);
5:        USE(C);
```

For the above program, no storage overlay is possible, and 70 units of storage are required. This situation can be improved by moving B into A before statement 4 and using A instead of B in statement 4. The resulting program is as follows:

```
        DEF A(20), B(20), C(30);
1:        SET(A,B);
2:        USE(A,B) GOTO(3,4a);
3:           SET(C) USE(A) GOTO(5);
4a:            SET(A) USE(B);          /* A = B */
4:             SET(C) USE(A);
5:        USE(C);
```

Now, C and B can share storage. The resulting storage assignment is as follows:



Total: 50

In Chapter 4, repositioning and unsharing will be formulated as a single, renaming problem. A canonical renaming transformation will be presented, and it will be shown that this transformation always reduces the maximum clique set size to the maximum active set size.

2.4 Code-Modifying Transformations

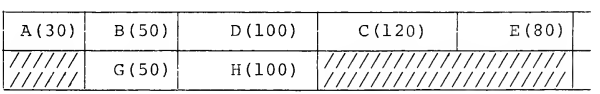
In the area of time optimization, certain code transformations have been found to improve object execution time. Many of these transformations, their inverses, or their generalizations, are also applicable to storage optimization, because they expose new opportunities for data overlay. In some cases, these transformations coincide with techniques used by storage-optimizing programmers. Some examples are given below.

2.4.1 Redundant Code Insertion

There are circumstances when storage can be saved by recalculating a set of values, instead of using the storage to hold the values between nonadjacent uses. The following example illustrates this point:

```
DEF A(30), B(50), C(120), D(100), E(80), G(50), H(100);
   SET(A,B);
   SET(C) USE(A,B);
   SET(D) USE(A,C);
   SET(E) USE(A,B);
   USE(D,E);
   SET(H) USE(C,A);
   SET(G) USE(C,A);
   USE(G,H);
```

The following storage layout is best for this program:

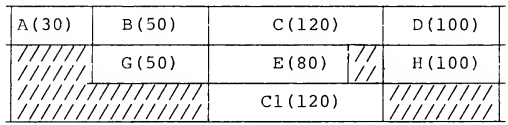


Total: 380

By recalculating C and using a renamed variable for the result, we get the following program:

```
     DEF A(30),B(50),C(120),Cl(120),D(100),E(80),G(50),H(100);
     SET(A,B);
1:     SET(C) USE(A,B);
     SET(D) USE(A,C);
     SET(E) USE(A,B);
     USE(D,E);
     SET(Cl) USE(A,B);    /* this is a copy of statement 1 */
     SET(H) USE(Cl,A);
     SET(G) USE(Cl,A);
     USE(G,H);
```

Now, the following assignment is possible, saving 70 units of object storage, at the expense of instruction storage and program execution time.



Total: 310

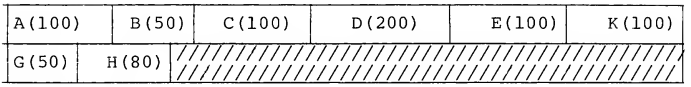
Redundant code insertion will be discussed in Chapter 5 of this paper.

2.4.2 Code Motion

Code motion can improve data space utilization. Consider the following example:

```
     DEF A(100),B(50),C(100),D(200),E(100),G(50),H(80),K(100);
1:   SET(A,B,C);
2:   SET(D) USE(A,B);
3:   SET(K) USE(A,C);
4:   SET(E) USE(A,D);
5:   SET(G) USE(B,K,E);
6:   SET(H) USE(C,D,K);
7:   USE(G,H);
```

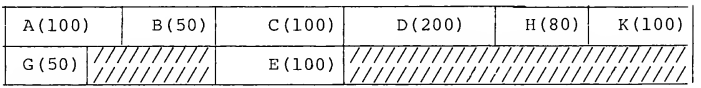
The following storage assignment is the best that can be done for this program as it stands:



Total: 650

If statement 6 is moved up two statements, the improved overlay of storage indicated below is possible, resulting in a 20-unit savings:

```
     DEF A(100),B(50),C(100),D(200),E(100),G(50),H(80),K(100);
 1:  SET(A,B,C);
 2:  SET(D) USE(A,B);
 3:  SET(K) USE(A,C);
 4:  SET(H) USE(C,D,K);
 5:  SET(E) USE(A,D);
 6:  SET(G) USE(B,K,E);
 7:  USE(G,H);
```



Total: 630

Moving the statement has changed the live conflicts of the program in such a way that the amount of overlayable storage is increased. Code motion will be discussed in Chapter 5 of this paper.

2.4.3 Splitting

Loop splitting, a generalization of the "unswitching" transformation discussed in the Cocke-Allen catalog (AlC72a), is useful in exposing opportunities for redundant code insertion and code motion. In the following example, DO and END statements delimit an iterative loop.

```
     DEF A(100), B(100);
     SET(A,B);
     SET(SUMA,SUMB);
LP:    DO USE(I) SET(I) TEST;
          SET(SUMB) USE(SUMB,I,B);
          END LP;
     USE(SUMA,SUMB);
```

This program requires 202 units, and no overlay is possible. However, if the loop and the first two nodes are split, the code can be reordered so that the storage requirement can be cut in half:

```
     DEF A(100), B(100);
     SET(A);
     SET(SUMA);
LP:    DO USE(I) SET(I) TEST;
          SET(SUMA) USE(SUMA,I,A);
          END LP;
     USE(SUMA);
     SET(B);
LP1:   DO USE(I1) SET(I1) TEST;
          SET(SUMB) USE(SUMB,I1,B);
          END LP1;
     USE(SUMB);
```

Now A can share storage with B, as can SUMA with SUMB and I with I1. The data storage savings must be offset, of course, against the increased instruction storage. Splitting will be discussed in Chapter 5 of this paper. In Chapter 6, this example will be used to illustrate the further improvement in storage utilization that is possible if a rank reduction transformation is performed.

Thus, we have seen several examples of code transformations that enhance the opportunities for automatic overlay of data. These transformations and others will be discussed in Chapters 4-6 of this paper.

3. AUTOMATIC DATA OVERLAY

In Section 2.1, an example of data overlay was presented, and it was observed that the key to automatic data overlay is the determination of each program variable's active sets. In this chapter, we shall consider the relationship of automatic data overlay to other known problems, and explore various approaches to a heuristic for the problem.

In Section 3.1, the data overlay problem is formulated as an extended coloring problem, and an exponential algorithm for finding a minimum solution is presented. Array data overlay for a straight line program is equivalent to the Shipbuilding Problem, and scalar data overlay for a general program is equivalent to the Graph Coloring problem, as well to a Renaming Problem. All three of these problems are NP-complete. In the succeeding three sections, these formulations are discussed.

In Section 3.5, the SOCRATES family of automatic data overlay algorithms is described. In Section 3.6, an approximation algorithm for automatic data overlay due to A. Hoffman is presented, together with his proof that the algorithm always realizes a bounded result. This result is important because it extends the known bound for scalar coloring, and because it provides a target for storage-optimizing code transformation.

In Section 3.7 further considerations in heuristic design are discussed.

3.1 An Exact Overlay Algorithm

The definitions of coloring and chromatic number can be extended so that a coloring corresponds to a storage layout and the chromatic number corresponds to the minimum size of a storage layout. For each node IV of an array conflict graph, a coloring COL maps IV into a pair of integers

COL: IV -> (A,S)

where S = |IV|, and the following condition holds:

If (IV,IV1) is in CG then either:

(i) COL(IV)(1) + COL(IV)(2) < COL(IV1)(1); or

(ii) COL(IV1)(1)+ COL(IV1)(2)< COL(IV)(1)

Thus, the color of an array is an interval along the positive integer axis whose size is the array size, and the colors of conflicting variables do not overlap.

For a particular coloring COL, the value of the coloring, ROOM, is defined as:

MAX /<< COL(IV)(1) + COL(IV)(2) | IV IN VARS >>

The extended chromatic number CHR of an array conflict graph is defined as the minimum value of ROOM over all possible colorings COL.

To find CHR for a particular conflict gragh, all valid layouts of unequal size must be explored. In the following algorithm, all possible orderings of VARS are examined, and, for each ordering, all possible positions at the left end of a preceding variable, or at the rightmost current storage position, are examined for each variable in turn. This generates all layouts, up to size equality. The algorithm is as follows:

```
ROOM := 1;  N = #VARS;  SPC := NL;
   FOR I  := 1 ... N DO
   VEC(I) := I;
   ROOM   := ROOM + |VARS(I)|;
   END FOR I;

   FOR I  := 1 ... FACT(N) DO
      EXPLORE(I,SPC,MEMS);
   END FOR I;

REPORT:  PRINT RSPC, ROOM;
      PROC EXPLORE(IV,SPC,MEMS);
      VAR I,J,SEG,V,SV,NS;
      IF IV > N
           THEN RSPC := SPC;
                ROOM  := NEXT(SPC(#SPC));
                IF ROOM= MAXCLIQ
                     THEN GOTO REPORT;  ENDIF;
           ELSE V := VARS(VEC(IV));
                SV := SZ(V);  NS = #SPC;
                     FOR I := 1 ... NS-1 DO
                     SEG   := 0;
                          FOR J := I+1...NS DO
                          PR := SPC(J);
                               FOR ALL X IN MEMS(PR) DO
                               IF(V,X) IN CG(2)
                                    THEN CONTINUE FOR I;
                               END FOR ALL;
                          SEG := SEG + PR(2);
                          END FOR J;
                     TRY(V,SV,IV,I,J,SPC,MEMS,SEG);
                     END FOR I;
                TRY(V,SV,IV,NS+1,NS+1,SPC,MEMS,0);
                END IF IV;
           END EXPLORE;

     PROC NEXT(PR);
     RETURN(PR(1) + PR(2));
     END NEXT;

           PROC TRY(V,SV,IV,I,J,SPC,MEMS,SEG);
           LOCAL NUSPC, NUMEMS;
           DIFF := SEG - SV;
           DEL := (DIFF > 0);
           K := 1;   M := J;
                FOR L := 0 TO DEL DO
                     FOR K := K...M DO
                     NUSPC(K+L) := SPC(K);
                     (FORALL W IN SPC(K))
                          INMEMS(K+L,W);  END FORALL;
                     END FOR K;
                M = #SPC;
                END FOR L;

           IF DIFF > 0
                THEN NUSPC(J)(2) := NUSPC(J)(2) - DIFF;
                     NUSPC(J+1)  := (NEXT(NUSPC(J),DIFF);
                     (FORALL W IN SPC(J)) INMEMS(J+1,W);
                ENDIF;
```

```
        IF DIFF < 0
             THEN NUSPC(M)(2) := NUSPC(M)(2) - SEG;
                  NUSPC(M+1)  := NEXT(NUSPC(M),-DIFF);
                  IF NEXT(NUSPC(#NUSPC+1)) >= ROOM
                       THEN RETURN;
             END IF DIFF;
        (FOR K := I...J + DEL)
             INMEMS(K,V);  END FOR K;
        EXPLORE(IV+1,NUSPC,NUMEMS);
        END TRY;
   PROC INMEMS(I,X);
   (NUSPC(I),X) IN NUMEMS;
   END INMEMS;
```

Each enumerated layout (duplicates are possible) is examined to see whether it is minimum. However, in actuality, since a running minimum ROOM is maintained in the program, layouts are not enumerated once their length exceeds the current value of ROOM, in the interest of efficiency. Also, once ROOM = MAXCLIQ, the algorithm terminates, since no shorter layout is possible.

EXPLORE tries to overlay the IVth variable V at the left end of each of the segments I in the current layout SPC by verifying that V does not conflict with any of the current variables that would be overlaid if V were positioned there. Each variable's boundary defines a new segment. For every valid positioning of V, TRY is invoked to produce a new copy of SPC and MEMS, and a recursive invocation of EXPLORE examines all possible positionings of the (IV + 1)st variable.

GENPERM generates successive permutations of the first N integers, so that EXPLORE can be attempted for all permutations of variable orderings. Chapter 1 of reference (Ev) contains several such algorithms. It may be noted that the execution time of the overlay algorithm can be reduced further if, in TRY, when the IVth variable is rejected, information is transmitted to GENPERM to bypass all permutations with the current prefix.

3.2 A Shipbuilding Problem

Consider the automatic overlay problem when the program is, or is approximated by, a straight-line sequence of instructions. At each statement I, let CRE(I) denote the set of variables "created" at statement I; that is, those variables that are active at statement I, but not at any statement J < I. At each statement I, let REL(I) denote the set of variables released at statement I; that is, those variables that are active at statement I, but not at any statement K > I. Consider object storage to be modeled as an interval along the integer axis.

A compiler progressing along the program, statement-by-statement, in execution order, can perform data overlay as follows: At each statement I, the variables in CRE(I) are assigned storage locations. Then, each variable in REL(I) can relinquish its current storage location, so the program simulates the freeing of the storage. The process is repeated for each statement. The problem is to assign the locations in such a fasion that the overall amount of storage required is minimized. This view of automatic data overlay is equivalent to the Shipbuilding Problem.*

In the Shipbuilding Problem, ships arrive and depart at specified times for servicing at a pier. Each ship's size is specified, and the problem is to minimize the maximum amount of pier space needed at any point in time. Thus, the ships correspond to program variables, the pier to data storage, the set of ships arriving at time I to CRE(I), the set of ships leaving at time I to REL(I), and the size of a ship to |V|. This problem has been shown to be NP-complete in work by A. Hoffman,** E. Johnson,** L. Stockmeyer,** and M. Golumbic.*** In their work, the following theorem is proved:†

---

* See reference (Go79), Chapter 9.

** IBM Research, Yorktown Heights, Unpublished results communicated in private conversation.

***Courant Institute of Mathematical Sciences, New York Univ.

† See reference (Go79), Chapter 9.

CHR is equal to the minimum over all directed acyclic orientations of CG of the length of the longest path.

A backtracking heuristic that exploits this theorem has been developed.

If SOCRATES produces evidence that the active patterns of program variables are close to single intervals, then the design of a Shipbuilding approximation algorithm should be addressed. We shall discuss this further in Section 3.6.

3.3 A Coloring Problem and Yershov's Heuristic

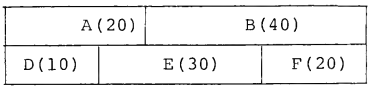
Yershov (Y71) has approached the problem of overlay determination as an extended graph coloring problem, using the conflict graph, as defined in Section 2.4.

Yershov observes that if all variables had the same size, the problem of overlay determination would be exactly equivalent to a scalar graph-coloring problem, and the effort would reduce to finding an effective coloring heuristic. Since variables of differing sizes are involved, a secondary heuristic which performs something of a packing function, must be integrated with the coloring heuristic.

In order to justify a combined approach, Yershov postulates a "Principle of Uniformity" that asserts that most programs have a uniform set of data variables — hundreds of scalars, say, but less than 100 arrays, and arrays that occur in small groups of approximately equal size. His approach is then to overlay variables of the same weight as much as possible, using the coloring heuristic, and then "pack" bigger arrays with smaller ones. The assumption is that it doesn't pay to cross array boundaries in the overlay process.

His approach has a number of shortcomings. Yershov's particular coloring heuristic can be arbitrarily bad for many conflict graphs. In fact, Garey and Johnson (GaJ76) have shown that any coloring heuristic can produce arbitrarily poor results on some graphs, so unless program conflict graphs have special properties that make them conducive to a particular heuristic, a coloring approach is not fruitful.

Another drawback is that, if the Principle of Uniformity does not apply, storage utilization can be quite poor. Overlay can take place only at the boundary of an overlaying variable, wasting available storage. For example, the following overlay structure could not be produced by Yershov's heuristic:



If D can only overlay A, and F can only overlay B, but E E can overlay both A and B, then Yershov's approach will result in a waste of at least 20 units.

3.4 A Renaming Problem

A more recent effort in overlay determination is that of Logrippo's (Log78). His approach is to treat overlay determination as a renaming problem, and he claims that this approach yields better results than Yershov's in providing for "one instance" of economy of memory, although he does not substantiate his claim in the paper. Since the size of a variable is not mentioned, it is difficult to understand how packing is to take place.

It does not seem that the renaming approach can provide an improved solution in the general case. If repeated name merges of variables are to be performed until the conflict graph becomes a clique (as Logrippo's paper suggests), then no packing is performed at all, with results vastly inferior to Yershov's. If some provision is made for packing, then overlay opportunities are still overlooked; for example, the overlay structure pictured in the previous section could not have been produced if the requirement for array D had occurred in a program state prior to the requirement for array E. In other words, overlay choices are made only on the basis of state proximity without considering size matching or number of overlay alternatives. Thus, a heuristic based on a renaming approach appears unattractive.

The view of renaming in this paper is at the opposite pole to Logrippo's view. Where Logrippo identifies the name of a variable and its storage, the present approach separates the storage from the variable. In Chapter 4, the canonical renaming transformation will introduce new variables instead of merging nonconflicting ones.

3.5 Heuristics for Automatic Data Overlay

John Cocke* has proposed an approach to conflict graph coloring, based on a backtracking algorithm devised by Ashok Chandra* and adapted by Gregory Chaitin* for register allocation. The heuristic is so good for scalar conflict graphs that backtracking to recolor nodes is almost never needed, and MAXCLIQ colors are almost always sufficient. This suggests that the chromatic number of a scalar conflict graph is usually MAXCLIQ, and that conflict graphs may have other properties that make them amenable to this heuristic.

Using MAXCLIQ as a starting value for NCOLS, the number of available colors, the original algorithm used a figure of merit to select nodes to be colored that varied directly as the number of uncolored neighbors and indirectly as the number of available colors. If, for any node, the number of available colors ever reached zero, NCOLS was increased by one.

This approach to coloring can be generalized in a number of ways for the extended coloring problem. In SOCRATES, the following method has been used, producing a family of approximation algorithms:

Let ROOM, the overlay storage requirement determined by the heuristic, be initialized to |VARS|. A storage segment is a pair (A,S), where A is the address of the leftmost location of the segment and S is the segment size. The overlay algorithm proceeds by selecting for each variable X in turn, a storage segment (A,S) out of AVSTO(X) in a sequence determined by SUMAV(X) and DGREE(X), where AVSTO(X), SUMAV(X) and DGREE(X), are defined as follows:

* IBM Research, Yorktown Heights, unpublished work communicated privately.

```
AVSTO(X) := <<(A,S) | A=1...ROOM AND S> = |X|
                AND FORALL(X,Y) IN CG  |
                (LOC(Y) + |Y|<= A)  OR
                (LOC(Y) >= A+S)     AND
                NOT EXISTS (A',S') IN AVSTO(X)
                | (A' <= A AND A'+ S' >= A+S) >>;
SUMAV(X) := (+ / << S | (A,S) IN AVSTO(X)>>) - |X|;
DGREE(X) := + / << |Y| | (X,Y) IN CG AND LOC(Y) = 0 >>;
```

The algorithm assigns X to the leftmost portion of the leftmost element of AVSTO(X), and LOC(X) is set to A.

Observe the following:

(a) The smaller SUMAV(X) is, the less freedom of choice there is for a location for X, and, therefore, the more urgent it is to assign X next.

(b) The larger DGREE(X) is, the more the total sum of conflicts involving X there are, and, hence, the more likely it is that by assigning X next, a greater quantity of overlay opportunities will materialize for the remaining unassigned variables.

The overlay algorithm begins by initializing AVSTO(IV) to (1,|VARS|) for all variables IV; that is, a one-element available storage list AVSTO(IV) is created for each IV. SUMAV(IV) and DGREE(IV) are initialized for each variable, in accordance with the above formulas.

The main loop is repeated until all variables have been assigned. First, SUMAV(IV) and DGREE(IV) are examined for each variable, seeking IMF, the index of a variable chosen according to one of the following possible sequencing rules:

Ascending Available Storage

Let IMF be the index of the variable with the smallest SUMAV(IV); if there is more than one variable with SUMAV(IV) equal to the smallest SUMAV(IV), let IMF be the index of the one with the largest DGREE(IV).

Descending Degree
Let IMF be the index of the variable with the largest DGREE(IV); if there is more than one variable with DGREE(IV) equal to the largest DGREE(IV), let IMF be the index of the one with the smallest SUMAV(IV).

Descending Figure of Merit
Let IMF be the index of the variable with the largest DGREE(IV) / SUMAV(IV).

Descending Weighted Figure of Merit
Let IMF be the index of the variable with the largest DGREE(IV)**W1 / SUMAV(IV)**W2, where W1 and W2 are some given weights.

At the end of the search, variable IMF has been selected for assignment. This is performed on a first-fit basis; that is, the variable is assigned storage from the leftmost portion of its leftmost AVSTO list element. The assigned AVSTO element is deleted from the AVSTO list of all conflicting variables. (These lists are maintained in such a way that the segment is ignored if its size drops below the size of the variable during this process.) IMF's available storage list is freed, and NN, the number of variables remaining to be assigned is decremented by 1.

In Section 8.1, we report on tests that were performed using the SOCRATES overlay procedures to evaluate some of these heuristics (Ascending Available Storage, Descending Degree and Descending Figure of Merit), as well as the algorithm described in the next section.

3.6 A Bounded Approximation Algorithm

In this section, we consider the question of an upper bound for the extended chromatic number. We present an algorithm and proof, due to A. Hoffman,* that demonstrates such a bound.

* Unpublished result communicated during discussions.

In the scalar case, Brooks (Hara) has demonstrated an upper bound of 1 + MAXD for the chromatic number, where MAXD is the maximum of DEGR(V), the degree of a node V, over all nodes of the graph. This result was sharpened by Szekeres and Wilf (SzWi68), by replacing MAXD with: the maximum over all subgraphs G' of the graph of:

```
MIN / << DEGR(V) | V IN G' >>
```

Their proof does not generalize for the extended coloring problem. However, an alternate proof, due to A. Hoffman, is applicable in both cases. Define the NVARS by NVARS symmetric array TERM as follows:

```
IF I = J
     THEN TERM(I,J) := |V(I)|;
     ELSE IF (I,J) IN CG(2)
          THEN TERM(I,J) := |V(I)| + |V(J)| - 1
          ELSE TERM(I,J) := 0;
```

For each row I, define:

```
ROWSUM(I) := (+ / <<TERM(I,J)|J=1...NVARS>>)
```

Theorem. An upper bound for the extended chromatic number CHR is given by the maximum over all principal submatrices T of TERM of the minimum sum of any row in T; i.e.,

```
CRITRSUM := (MAX /
              <<MIN /
                 <<ROWSUM(I) | I IN T >>
                             | ISSUBMAT(T,TERM) >>
```

Proof: Consider the following algorithm:

Step 1. Build a list of variables to be assigned as follows; with K initially equal to NVARS.

(a) Find the row I in TERM with the smallest rowsum.

(b) Insert V(I) in the Kth slot of the assignment list, and decrement K by 1.

(c) Delete row I and column I from TERM.

(d) Repeat above steps NVARS times.

Step 2. Assign storage in assignment list order, choosing any available (nonconflicting) storage segment for each variable.

We shall show by induction on NVARS that the storage required is bounded by CRITRSUM. For NVARS = 1, the claim is true. Assume it is true for NVARS < k.

The inductive assumption implies that all the variables assigned storage in Step 2, except the last, fit into a storage segment whose size is less than or equal to CRITRSUM. Suppose that there is no room for V(I), the kth variable in the assignment list. Let V(J(1)),...,V(J(P)) denote the variables conflicting with V(I), in increasing assigned storage order. Suppose that there are Q "gaps" in the storage, where S(1) denotes the size of the 1-th gap. If V(I) does not fit, we must have the following inequalities:

$$|V(I)| >= S(1) + 1$$
$$|V(I)| >= S(2) + 1$$
$$\cdots$$
$$|V(I)| >= S(Q) + 1$$

That is,

$$Q * (|V(I)| - 1) + |V(J(1))| + \cdots + |V(J(P))| >=$$
$$S(1) + \cdots + S(Q) + |V(J(1))| + \cdots + |V(J(P))|$$

Therefore, since Q <= P+1,

$$ROWSUM(I) - 1 >= CRITRSUM$$

This is a contradiction of the definition of CRITRSUM.

This result is significant for the extended coloring problem, because now heuristics can be aimed at transforming a program so as to reduce MAXCLIQ and/or CRITRSUM.

The above algorithm, with a first-fit assignment strategy, has been added to the SOCRATES overlay algorithms, and experimental results on its effectiveness are reported in Section 8.1.

3.7 Design Implications

The result of Garey and Johnson (GaJ76) ensures that the algorithms in the preceding sections will produce sub-optimum results on some graphs. One purpose of the SOCRATES study is to see whether array conflict graphs have properties that make them amenable to one of these heuristics. The hypothesis is that programs do not generate pathological conflict graphs very often.

SOCRATES is also investigating the active interval distributions so that evidence can be compiled on the probable efficacy of a heuristic that performs particularly well on shipbuilding graphs. It should be observed that shipbuilding graphs have a distinctive structure: they consist of a succession of mutually overlapping cliques ACT(I), with the property that, if some variable V is not in the intersection of ACT(I - 1) and ACT(I), then it is not in any ACT(J), J > I. We conjecture that this structure can be exploited to produce an effective approximation algorithm for the Shipbuilding problem. (It is not known whether the Garey-Johnson result holds for the Shipbuilding problem.)

In the following chapters, storage-optimizing transformations will be described. These can be applied to reduce MAXCLIQ and/or CRITRSUM. The question of whether these transformations actually improve the results of some overlay heuristic is problematical, since there is no guarantee that reducing the upper bound and/or the lower bound will reduce the chromatic number, let alone the overlay algorithm result.

Pragmatically speaking, this lack of a guarantee is not critical for a storage-optimizing compiler, since the results of the overlay heuristic beofre and after the transformation can be compared by the compiler. In many instances, the conflicts between a troublesome pair of variables can be eliminated by another transformation, so that the overlay results can be improved heuristically.

4. THE RENAMING TRANSFORMATIONS

Renaming transformations modify the program by introducing new variables to assume some of the conflicts of old variables, so that cliques in the conflict graph are broken. These transformations are applicable when MAXLIVE < MAXCLIQ. In this chapter, it will be demonstrated that MAXCLIQ can always be reduced to MAXLIVE by a compound renaming transformation. Renaming transformations can also be used to break up cliques that contribute to CRITRSUM.

This result is significant for register optimization as well as storage optimization. Register optimization corresponds to a scalar conflict graph coloring problem. The canonical renaming transformation will reduce a scalar conflict graph's maximum clique to the scalar MAXLIVE, which is the maximum number of registers needed at any particular node of the program; the algorithm will determine exactly where in the program the "move-register" operations needed to effect this reduction should be inserted.

In the preceding section, we pointed out that it is an open question whether reducing MAXCLIQ will guarantee overlay result improvement, although, if a counterexample should be demonstrated in the future, a poor overlay result can often be improved heuristically by storage-optimizing code transformation. Examples of programs where renaming improves storage utilization are plentiful, and neither imagination nor empirical evidence has yet produced a counterexample. Thus, it appears that renaming transformations will be an important part of a storage-optimizing compiler.

In Section 2.3, examples of unsharing and repositioning were given. In Section 4.1, these transformations are introduced, and in Section 4.2 they are specified in detail and their implementation in SOCRATES is described. The canonical renaming transformation is specified in Section 4.3, and examples of its application are presented in Section 4.4.

4.1 Background

4.1.1 Unsharing

Suppose X is a variable in some clique C, |C| = MAXCLIQ, and suppose X is active at nodes J1,...,Jk of ND. Consider the nodes LNACT(X) = <<J1,...,Jk>>. Unsharing effects a partitioning of these nodes into one or more disjoint subsets under the equivalence relation:

Jm .eq. Jn iff Jm IN SUCCS(Jn) OR Jn IN SUCCS(Jm)

Each equivalence class identifies another "name" of X. Unsharing consists of discovering these "p names of X", and renaming X accordingly. In effect, the programmer has used one variable for p purposes, in a possibly misguided attempt to save storage. By applying the unsharing transformation, more overlay possibilities are introduced so that the overlay heuristic can yield better results.

4.1.2 Repositioning

Repositioning is applicable when the aggregation of different intervariable conflicts at several nodes has produced a greater intervariable conflict. For example, if X conflicts with Y at one node, and with Z at another, and if Y and Z conflict at a third node, then X, Y and Z are mutually conflicting variables, although the entire conflict is not active at a single node. As another example, if the following active sets are present:

<<A,B,C,F>>, <<B,D,E,G>>, <<A,D,F>>,<<A,C,E,F>>

the following clique will be formed:

<<A,B,C,D,E,F>>

In both examples, a large clique is formed from smaller ones, and, as a result, MAXCLIQ may exceed MAXLIVE.

Suppose that (X,Y) are a pair of variables that both belong to some clique C in CS, where |C| = MAXCLIQ, and MAXCLIQ > MAXLIVE. Assume, without loss of generality, that |X| <= |Y|. Define the following sets:

```
BOTH   := LNACT(X) * LNACT(Y);
NBOTH  := LNACT(X) - LNACT(Y);
INEDGS := <<(M,N) IN FG(P) | M IN NBOTH AND N IN BOTH>>;
OUEDGS := <<(M,N) IN FG(P)|M IN BOTH  AND N IN NBOTH>>;
```

If NBOTH is nonempty, the repositioning transformation applies. A new variable, X_Y, is introduced, with |X_Y| = |X|. The program is modified as follows:

Step 1. FORALL N IN BOTH(X,Y):
Replace every reference to X by a reference to X_Y.
Step 2. FORALL(M,N) IN INEDGS(X,Y):
Insert a node between M and N consisting of the statement: X_Y := X;
Step 3: FORALL(M,N) IN OUEDGS(X,Y):
Insert a node between M and N consisting of the statement: X := X_Y;

For example, consider this program segment from Section 2.3.2:

```
      DEF A(20), B(20), C(30);
1:    SET(A,B);
2:    USE(A,B) GOTO(3,4);
3:      SET(C) USE(A) GOTO(5);
4:      SET(C) USE(B);
5:    USE(C);
```

For this program, the conflict graph is a triangle with vertices A, B and C, and with MAXCLIQ = 70, and MAXLIVE = 50. If repositioning is applied for the pair (B,C), the following sets are built:

```
BOTH   = <<4>>
NBOTH  = <<1,2>>
INEDGS = <<(2,4)>>
OUEDGS = << >>
```

The following program results:

```
      DEF A(20), B(20), C(30), B_C(20);
 1:     SET(A,B);
 2:     USE(A,B) GOTO(3,4a);
 3:       SET(C) USE(A) GOTO(5);
4a:         SET(B_C) USE(B);      /* B_C := B */
 4:       SET(C) USE(B_C);
 5:     USE(C);
```

Now, the conflict graph consists of the edges (A,B), (A,C), (B_C,B), and (B_C,C). These are also the cliques, and MAXCLIQ has been reduced to MAXLIVE. Observe that B_C can be overlaid on A and B can be overlaid on C.

4.2 SOCRATES Renaming Implementations

As will be demonstrated, a single transformation can effect both unsharing and repositioning. However, in the initial design stages of SOCRATES this was undiscovered, so separate procedures were implemented. Each procedure receives as argument VSETMSK, a set of variables to be renamed that is usually, but not necessarily, the maximum clique set. Thus, SOCRATES can be used to deliver information on the possible utility of eliminating noncritical edges from the conflict graph (i.e., edges not in a maximum clique set). SOCRATES may also yield data on whether reducing MAXCLIQ ever yields worse overlay results.

4.2.1 Unsharing

As already observed, the problem of finding the "right number of names" for a variable V is an equivalence class computation suitable for a "UNION-FIND" algorithm. The SOCRATES unsharing implementation is a straightforward PL/1 transcription of the UNION-FIND algorithm presented in (AhHU74), Chapter 4.7, page 132. This algorithm executes O(n) UNION and FIND instructions in almost linear time.

For every variable V in VSETMSK, the unsharing procedure computes the naming equivalence classes of V as follows: Each node at which V is active is initialized to a unique name. Then, iterating in inverse depth-first order over the nodes I at which V is active, the following steps are executed: (1) Node I's current name, NAMEI, is found via FIND; (2) Each of the successors J of I are examined, and, if V is active at J, NAMEJ, the current name at J, is found via FIND, and UNION is invoked to merge the equivalence classes of NAMEI and NAMEJ. At the end of the inverse depth-first iteration, the number of names is ascertained in the REPORT procedure, and, if the number of names is greater than 1, each reference node's name is reported.

4.2.2 Repositioning

Each distinct pair (IX,IY) of variables in VSETMSK is examined. BOTH(IX,IY) and NBOTH(IX,IY) are calculated in accordance with the formulas of Section 4.1.2, and, if they are not empty, the pair (IZ,IZ1) is examined, where, if |IX| >= |IY|, then IZ = IX and IZ1 = IY, otherwise IZ= IY and IZ1 = IX. INEDGS(IZ,IZ1) and OUEDGS(IZ,IZ1) are calculated, as defined in 4.1. GOODPRT reports the success of the repositioning transformation for variables IZ and IZ1, printing out the substitutions for the BOTH set, and the insertions for INEDGS and OUEDGS.

4.3 A Canonical Renaming Transformation

In this section we shall prove that, if MAXCLIQ > MAXLIVE, then repeated applications of the repositioning transformation, defined in the preceding section, will eventually reduce MAXCLIQ.

Suppose C is a clique such that |C| = MAXCLIQ > MAXLIVE. Define the following sets:

```
ANYACTND = { / ·· LNACT(X) * LNACT(Y) |
             X IN C AND Y IN C AND X NOT = Y ··>;
ALLACT   =  ··X IN C|(FORALL I IN ANYACTND) X IN ACT(I) ··;
SOMACT   = C - ALLACT;
```

Note that 1 <= #COMPACT <= #C;

Suppose that COMPACT = Y(1),...,Y(k), where |Y(1)| <= |Y(i)|, i = 2,...,k. We define the following notation:

X(1) = Y(1)

X(i+1) = X(i) Y(i+1)

That is, X(i+1) is the renamed variable obtained at the ith application of repositioning. We make the following modification in the definition of repositioning for the purpose of this renaming transformation:

> At the ith application of repositioning, if a node 1 of the form:
>
> X(j) : X(i - 1)
>
> is in BOTH(X(j), X(i + 1)), then, instead of the substitution of X(i + 1) for X(j) which ordinarily would have been performed, insert, instead, a node of the form:
>
> X(i + 1) : X(j)
>
> immediately after node 1. Similarly, if a node 1 of the form
>
> X(i - 1) : X(j)
>
> is in BOTH(X(j),X(i+1)), then, instead of the substitution of X(i+1) for X(j) which ordinarily would have been performed, insert, instead, a node of the form:
>
> X(j) : X(i + 1)
>
> immediately before node 1. Then the following inductive claims hold at the ith application of repositioning:
>
> Claim 1. The semantics of the program is unaffected (up to interruption).
>
> Clearly, the movement of data between old and new variables preserves the value necessary to retain the semantics of the program.
>
> Claim 2. MAXLIVE is not increased.
>
> Consider each step in turn:

Step 1. FORALL N IN BOTH(X(i),Y(i+1)),|ACT(N)| is unaffected by the substitution.

Step 2. FORALL(M,N) IN INEDGS(X(i),Y(i+1)), let I denote the inserted node:

$$ACT(I) = ACT(M) * ACT(N) + \langle\langle X(i+1) \rangle\rangle$$

$$|ACT(I)| = |ACT(M) * ACT(N)| + |X(i+1)|$$

$$|ACT(I)| \leq (MAXLIVE - |Y(i+1)|) + |X(1)|$$

$$|ACT(I)| \leq MAXLIVE$$

Step 3. FORALL(M,N) IN OUEDGS(X(i),Y(i+1)), let I denote the inserted node:

Proof similar to step 2.

Claim 3. No conflict graph contains a clique that includes more than two renamed variables X(j-1) and X(j).

At each original program node, only one X(j) can be active. At the inserted nodes, the renaming transformation has been defined in such a way that only a pair of renamed variables (X(j-1), X(j)) are active at any node.

Claim 4. After the ith repositioning, the only possible cliques of size MAXCLIQ are maximum-sized cliques other than C in the original graph or a clique containing X(i+1).

Let CG(i) denote the conflict graph after the ith application of repositioning. In the transition from CG(i-1) to CG(i), cliques without X(i) are unaffected, and cliques in CG(i-1) with X(i) but without Y(i+1) are unaffected in size. Thus, the only possible cliques of size MAXCLIQ in CG(i) correspond either to maximum-sized cliques other than C in the original graph or to cliques in CG(i-1) containing both X(i) and Y(i+1). The latter correspond to cliques in CG(i) containing X(i+1).

We now state the following lemma.

Lemma. Suppose again that C is the only clique in the program of size MAXCLIQ, MAXCLIQ > MAXLIVE, and suppose that SOMACT = <<Y(1),...,Y(k)>>, where |Y(1)| <= |Y(i)|, i = 2,...,k. Suppose that the renaming transformation is applied as above, using at most (k-2) repositioning applications. Then, the size of the maximum clique in CG(k-2) is strictly less than MAXCLIQ.

Proof: Suppose this were not so. Then, by Claim 4 above, there would be a clique in CG(k-2) containing X(k-1) and Y(k). But this would mean that, in the original program, there was some node at which Y(1),...,Y(k) were all active, contradicting the definition of SOMACT.

We thus have the following theorem.

Theorem. If MAXCLIQ > MAXLIVE, then MAXCLIQ can be reduced to MAXLIVE by repeated applications of the repositioning transformation.

Proof: If C is the only clique whose size exceeds MAXCLIQ, then Lemma 2 can be applied repeatedly as long as MAXCLIQ > MAXLIVE; eventually MAXCLIQ must be reduced to MAXLIVE. If there is more than one such clique, then the process can be applied to each clique in turn.

Note that this transformation can also be applied to break up other cliques in the conflict graph. The transformation is applicable to any clique C as long as #SOMACT(C) > 1. In particular, CRITRSUM can be reduced by this technique.

4.4 Renaming Transformation Examples

In order to illustrate the renaming algorithm, let us consider a few more examples.

```
          Code                          Active Sets
     DEF A(25),B(20),C(35),D(40);
 1:  SET(A,D);                          <<A,D>>
 2:  SET(B) USE(A) GOTO(3,5,7);         <<A,B,D>>
 3:  SET(C) USE(A,B) GOTO(4,7)          <<A,B,C>>
 4:  SET(D) USE(A,C) GOTO(7);           <<A,C,D>>
 5:  SET(D) USE(A,B);                   <<A,B,D>>
 6:  SET(C) USE(A,D);                   <<A,C,D>>
 7:  USE(D);
```

The clique set is:

```
<< <<A, B, C, D>> >>
SOMACT = <<B,C,D>>
```

Since MAXCLIQ > MAXLIVE, we apply the renaming transformation for (B,C):

```
BOTH   = <<3>>
NBOTH  = <<2,5>>
INEDGS = <<(2,3)>>
OUEDGS = << >>
```

```
          Code                          Active Sets
     DEF A(25),B(20),B_C(20),C(35),D(40);
 1:  SET(A,D);                          <<A,D>>
 2:  SET(B) USE(A) GOTO(3a,5,7);        <<A,B,D>>
3a:  B_C := B;                          <<B,B_C,A>>
 3:  SET(C) USE(A,B_C) GOTO(4,7);       <<A,B_C,C>>
 4:  SET(D) USE(A,C);                   <<A,C,D>>
 5:  SET(D) USE(A,B);                   <<A,B,D>>
 6:  SET(C) USE(A,D);                   <<A,D>>
 7:  USE(A,D);
```

The clique set is:

```
<< <<A,B,D>>, <<A,C,D>>, <<A,B,B_C>>, <<A,B_C,C>>
```

Now, MAXCLIQ = MAXLIVE, and no further transformations are necessary.

Now consider the following program:

```
        Code                                Active Sets
    DEF A(25),B(30),C(35),D(20),E(10);
 1: SET(A,B,C) GOTO(2,3,4);                 <<A,B,C>>
 2: SET(D) USE(A,B) GOTO(5);                <<A,B,D>>
 3: SET(D) USE(A,C) GOTO(5);                <<A,C,D>>
 4: SET(D) USE(B,C);                        <<B,C,D>>
 5: SET(E) USE(D);                          <<E,D>>
```

Clique set:

```
<< <<A,B,C,D>>, <<D,E>>  >>
SOMACT = <<D,A,B,C>>
BOTH(D,A)  =  <<2,3>>
NBOTH      = <<4,5>>
INEDGS     = << >>
OUEDGS     = <<(2,5), (3,5)>>
```

```
     Transformed Code                       Active Sets
    DEF A(25),B(30),C(35),D(20),D_A(20);E(10);
 1: SET(A,B,C) GOTO(2,3,4);                 <<A,B,C>>
 2: SET(D_A) USE(A,B);                      <<A,B,D_A>>
21: D = D_A GOTO(5);                        <<D,D_A>>
 3: SET(D_A) USE(A,C);                      <<A,C,D_A>>
31: D = D_A GOTO(5);                        <<D,D_A>>
 4: SET(D) USE(B,C);                        <<B,C,D>>
 5: SET(E) USE(D);                          <<E,D>>
```

Clique set:

```
<<  <<A,B,C>>, <<D,E>>, <<D,D_A>>,
    <<B,C,D>>, <<A,B,C,D_A>>       >>
```

A second application is necessary:

```
BOTH(D_A,B) = <<2>>
NBOTH =  <<21,3,31>>
INEDGS  = << >>
OUEDGS  =  <<(2,21)>>
```

	Transformed Code	Active Sets
	DEF A(25),B(30),C(35),D(20),	
	D_A(20), D_A_B(20), E(10);	
1:	SET(A,B,C) GOTO(2,3,4);	<<A,B,C>>
2:	SET(D_A_B) USE(A,B);	<<A,B,D_A_B>>
21:	D_A = D_A_B;	<<D_A,D_A_B>>
22:	D = D_A GOTO(5);	<<D,D_A>>
3:	SET(D_A) USE(A,C);	<<A,C,D_A>>
31:	D = D_A GOTO(5);	<<D,D_A>>
4:	SET(D) USE(B,C);	<<B,C,D>>
5:	SET(E) USE(D);	<<E,D>>

Clique set:

<< <<A,B,C>>, <<D,E>>, <<D,D_A>>,
<<A,B,D_A_B>>, <<D_A, D_A_B>>,
<<B,C,D>>, <<A,C,D_A>> >>

5. BASIC CODE-MODIFYING TRANSFORMATIONS

Attention now turns to transformations that split, move or replicate the nodes of a program graph for the purpose of eliminating all active conflicts between a pair of variables, thus permitting the variables to share storage. The relationship of the code modification problem to the NP-complete register-minimizing code ordering problem is discussed. An overview of the approach taken in SOCRATES is described. The safety and profitability constraints for hoisting, sinking and copying are given, together with a description of the SOCRATES' implementation.

5.1 Background

Suppose we have a linear code segment to which some subset of variables is local. Let CRE(I) and REL(I) be the set of variables created and released at node I, as defined in Section 3.2. Let TEMP(I), the intersection of CRE(I) and REL(I), denote node I's temporary variables. Let SUEX(I) denote the amount of storage in use on exit from node I, and MSIN(I) denote the minimum amount of storage needed during node I. Then the following formulas hold for each node I in the linear code segment:

(i) SUEX(0) = SO,

SUEX(I) = SUEX(I - 1) + |CRE(I)| - |REL(I)|

(ii) MSIN(I) = MAX(SUEX(I - 1), SUEX(I)) + |TEMP(I)|

The code reordering problem is that of finding a "legal reordering" of the nodes that minimizes MAX(MSIN(I)) without changing the semantics of any assignments. If this problem is restricted to scalar or equal-sized variables, it is equivalent to the problem of minimizing the number of registers needed to compute a sequence of scalar assignment statements with possible common subexpressions, a problem which Sethi

has shown to be NP-complete (Se75).

We conjecture that the register optimization problem, extended to permit redundant computations as well as reordered ones, is still NP-complete. This question, however, is outside the scope of the current research.

Aho, Johnson & Ullman (AhJU76), have found the "dag" useful to define the notion of a "legal order". For our purposes, let the nodes of a dag DG correspond to the nodes of the program segment. An edge (I,J) exists if and only if node I uses a value defined in node J or I is an input (output) statement that must precede the input (output) statement J. A legal ordering of the nodes corresponds to a topological sort of the nodes of DG. Aho, Johnson and Ullman have studied heuristics for the scalar reordering problem.

5.2 Goal-Directed Code Modification

The exponential nature of the code modification problem can be reduced to manageable proportions if a particular pair of variables is chosen for conflict elimination. The pair may be selected in one of several ways:

(1) If the conflict graph's MAXCLIQ, the lower bound for the overlaid storage requirement, is too big, then code modification can be attempted for each pair of variables in the maximum clique set.

(2) If the overlay heuristic has yielded a suboptimal storage utilization, then the overlay results can be examined to pinpoint a pair of variables whose overlay will produce an immediate improvement.

(3) The varfible whose rowsum is CRITRSUM (see Section 3.6) can be examined, together with all its adjacent nodes, for means of reducing the upper bound CRITRSUM.

Three types of active conflicts can occur between a given pair of variables at any particular node:

(1) both variables live;

(2) one variable live, subreference of other used, defined or modified;

(3) subreference of each used, defined or modified.

A type (1) conflict is eliminated by the removal of all other conflicts. A type (3) conflict is eliminated by splitting the node into two nodes with a type (2) conflict. As the examples in Section 2 illustrated, type (2) conflicts can often be eliminated by code motion or code replication.

Since arrays are usually assigned values in loops, SOCRATES investigates the move or copy of loops as well as statements. Loops are defined as a basic construct in the Storage Optimization Language (see Section 7.3). In what follows, the term unit will be used to denote a loop or single statement.

The examples in Section 2 illustrated code modification of straight-line segments. Let us consider some of the problems that arise when more complex control flow is taken into account.

5.2.1 Hoisting and Sinking

When forked control structures are addressed, code transformations like hoisting and sinking (AlC72a) become appropriate. In the following example, the relative optimality of HOIST or SINK depends on the relative sizes of B, C and G:

```
SINK:
  SET(A,B,C);
  USE(TEST) GOTO(THEN,ELSE);
    THEN:
      SET(G) USE(A);
      SET(D) USE(A,C); /* this is the moving node */
      SEG(H) USE(D,G) GOTO(END);
    ELSE:
      SET(G) USE(B);
      SET(D) USE(A,C); /* this is the moving node */
      SET(H) USE(D,G);
END: USE(G,H);
```

```
HOISE:
  SET(A,B,C);
  SET(D) USE(A,C); /* this is the moving node */
  USE(TEST) GOTO(THEN,ELSE);
    THEN:
      SET(G) USE(A);
      SET(H) USE(D,G) GOTO(END);
    ELSE:
      SET(G) USE(B);
      SET(H) USE(D,G);
 END: USE(G,H);
```

If we assume that the last uses of B and X are as shown in the code segments, and if we ignore for the moment the cost of instruction space, then, with the following declaration, SINK is better:

```
DEF A(100), B(100), C(20), D(40), G(10), H(10);
HOIST:
```

A(100)	B(100)	C(20)	D(40)
H(10)	//////////////////////	G(10)	//////////

Total: 260

SINK:

A(100)	B(100)	C(20)	G(10)
H(10) ////////	D(40)	//////////////////////	

Total: 230

With the following declaration, HOIST is better:

```
DEF A(100), B(20), C(100), D(40), G(50), H(10);
HOIST:
```

A(100)	B(20)	C(100)
//////////////	H(10) ///	G(50) /////////

Total: 220

SINK:

A(100)	C(100)		D(40)	G(50)
//////////////	H(10)	/////////	B(20)	///////////

Total: 290

With the following declaration, both are equally good:

```
DEF A(20), B(20), C(20), D(100), G(100), H(100);
```

HOIST or SINK:

D(100)	G(100)	H(100)			
////////////////////////////		A(20)	B(20)	C(20)	///

Total: 300

5.2.2 Code Grouping Problems

Although moving or copying a single node can often be effective, many occasions arise when the move or copy of a collection of nodes — not necessarily consecutive — will effect a storage savings. In such cases, several transformations may have to be applied to reorder, split, copy or regroup them for the purpose of the final move or copy. Consider the following example:

```
    DEF A(100),B(50),C(100),D(200),E(100),G(50),H(80),K(100);
    SET(A,B,C);
    SET(D) USE(A,B);
3:  SET(E) USE(A,D);
    SET(K) USE(A,C);
    SET(G) USE(B,K,E);
6:  SET(H) USE(C,D,K);
    USE(G,H);
```

The best order for this code is the following, which is the the same as in the example in Section 2.3.4:

```
    DEF A(100),B(50),C(100),D(200),E(100),G(50),H(80),K(100);
    SET(A,B,C);
    SET(D) USE(A,B);
    SET(K) USE(A,C);
6:  SET(H) USE(C,D,K);
3:  SET(E) USE(A,D);
    SET(G) USE(B,K,E);
    USE(G,H);
```

For this transformation to take place, node 3 must be moved down one node, and node 6 must then be moved up two nodes. It should be noted in this regard that there may be instances of a reordering that cannot be achieved by compounding of move transformations because intermediate transformations do not meet the profitability criterion.

Groups of statements can be moved or copied by using dominance trees and strongly connected regions to define the grouping. This was not included in the initial SOCRATES implementation, but is a candidate for future effort.

5.2.3 Multisource Transformations

Another difficulty that is side-stepped in SOCRATES, but should be addressed in future work, concerns the movement or replication of several units at a single target unit. The question of how the units should be inserted to achieve a semantically correct transformation is pertinent.

We observe that there are several special cases whose treatment is straightforward, and postulate that the problem may have a general solution. One such instance occurs when the target unit is a successor of all the source units, and all units are in the same strongly connected region (or not in a strongly connected region at all); then bit variables can be set in the code at the source units, one per source unit, to record the progress of the flow of control during execution, and tests at the target units can preface the inserted unit so that the appropriate (moved or copied) unit

is executed. This is illustrated by the following example:

```
     SET(A,B,C,D,E,F);
     USE(A,B,C) GOTO(L1,L2);
L1:  SET(A,B) USE(C,D);
     SET(G) USE(E,D) GOTO(L3);
L2:  SET(B,A) USE(C,F);
     SET(G) USE(E,F);
L3:  USE(A,B,C);
     USE(G) SET(H);
     USE(H);
```

Transformed code might be the following:

```
     SET(A,B,C,D,E,F);
     USE(A,B,C) GOTO(L1,L2);
L1:  SET(A,B) USE(C,D);
     SET(BIT) GOTO(L3);  /* BIT = 1 */
L2:  SET(B,A) USE(C,F);
     SET(BIT);           /* BIT = 0 */
L3:  USE(A,B,C,BIT) GOTO(LBIT1,LBIT2);
LBIT1:  SET(G) USE(E,D) GOTO(LBIT3);
LBIT2:  SET(G) USE(E,F);
LBIT3:  USE(G) SET(H);
     USE(H);
```

This idea can be generalized for other control configurations, using arrays or lists of bits. Future study should investigate the extent of such generalizations.

Multisource hoists are valid only if defined values of each moved unit are either disjoint or redundant. SOCRATES permits multisource sinks or copies to a unit, but when a multisource hoist to a unit is attempted, the user is advised to split, reorder and regroup his nodes. Extending SOCRATES to recognize redundant expressions is left for future effort (see Section 6.3).

5.2.3 Cascading Transformations

Redundant expression insertion can involve a cascade of copies if the values used in the source node are dead at the target node with a different set of reaching computations. This situation introduces complexities into the redundant expression insertion process. For the purpose of simplicity, such cascading was not addressed in SOCRATES, and is a primary candidate for follow-on activity.

5.3 Safety and Profitability Constraints

In this section, we describe the redundancy equations that SOCRATES solves in order to ensure the safety of a transformation, and the tests that SOCRATES performs in order to determine the profitability of the transformation. Since hoisting, sinking and copying are so similar in these respects, they are described together. We are interested in sufficient conditions, and leave for future study the discovery of necessary and sufficient conditions.

We will be using the following definitions:

Define GEN to be the set of "rdef pairs" (R,N), where N is a program node, and R is in MODS(N).

Define PAVIN (alternatively, PAVOUT) to be the set of triples (R,N,M), where (R,N) is an element of GEN, and the reference R calculated at node N is possibly available on entry to (alternatively, on exit from) node M. In Chapter 7, a method for calculating these sets will be presented.

Define DKILL to be the set of triples (R,N,M), where (R,N) is an element of GEN, and the reference R calculated at node N is definitely killed at node M.

Define CJUSES(I) (or CJMODS(I)) to be, respectively, the set of references, or conjoints of references, used (or modified) at node I.

To describe concisely the conditions governing hoisting, sinking and copying, we introduce four predicates — IUSEVALSEQ(I,J), OUSEVALSEQ(I,J), IMODVALSEQ(I,J) and OMODVALSEQ(I,J). The I-predicates describe conditions prevailing on entrance to unit I, the O-predicates describe conditions prevailing on exit from unit I. The USE-predicates relate to variables used in unit J, the MOD-predicates relate to variables modified or defined in unit J. Specifically, the definitions are as follows:

IUSEVALSEQ(I,J) is true if, for every reference R in CJUSES(J), PAVIN(J) <<R>> = PAVIN(K) <<R>>;

OUSEVALSEQ(I,J) is true if, for every reference R in CJUSES(J), PAVOUT(J) <<R>> = PAVOUT(I) <<R>>;

IMODVALSEQ(I,J) is true if, for every reference R in CJMODS(J), R is not in ACT(I), or:

PAVIN(I) <<R>> = (PAVIN(I) - DKILL(J) + GEN(J))<<R>>

OMODVALSEQ(I,J) is true, if, for every reference R in CJMODS(J), R is not in ACT(I), or:

PAVOUT(I) <<R>> = (PAVOUT(I) - DKILL(J)+GEN(J))<<R>>

The safety conditions for copying are as follows:

(COPY-1): The copy of a node J past a node I which it currently precedes is safe if the following conditions are met:

(a.1) No reference (or conjoint of a reference) used in J is modified in I.

(a.2) No reference (or conjoint of a reference) modified in J is modified in I.

(b) There is no branch into I that augments the set of possibly available values: that is, OUSEVALSEQ(I,J) and OMODVALSEQ(I,J) must hold.

(COPY-2): A node can be copied past itself only if its uses and modifications are nonoverlapping.

(COPY-3): A node I can be copied after a node J which it ultimately (or immediately) precedes, if it can be copied after each node along every path connecting I and J.

The following equations express these conditions:

```
SAFE1(I) := <<J IN ND, J NOT = I |
              CJMODS(I) * CJMODS(J) = NL AND
              CJMODS(I) * CJUSES(J) = NL >>+
            <<I | CJMODS(I) * CJUSES(I) = NL >>;
SAFE2(I) := <<J IN ND |
              OUSEVALSEQ(I,J) AND
              OMODVALSEQ(I,J) ) ) >>;
INSTSAFE(I) := SAFE1(I) * SAFE2(I);
COPYSAFE(I) := << I | CJMODS(I) * CJUSES(I) = NL >>
              + INSTSAFE(I) *
              ( */<<COPYSAFE(J)|J IN PREDS(I)>>);
```

The safety conditions for sinking are as follows:

(SINK-1): The conditions for sinking a node J past a node I which it currently precedes are as follows:

(a.1) No reference (or conjoint of a reference) used in J is modifed in I.

(a.2) No reference (or conjoint of a reference) modified in J is modified in I.

(a.3) No reference (or conjoint of a reference) used in I is modified in J.

(b) There is no branch into I that augments the set of possibly available values; that is, OUSEVALSEQ(I,J) and OMODVALSEQ(I,J) hold.

(SINK-2) A node can be moved past itself.

(SINK-3) A node I can be sunk after a node J which it ultimately (or immediately) precedes, if it can be sunk after each node along every path connecting I and J.

The following equations express the safety conditions for sinking:

```
SAFE3(I)   :=  << J IN SAFE1(I)|CJUSES(I) * CJMODS(J)=NL>>
             + << I >>;
MVDNSAFE(I) := SAFE3(I) * SAFE2(I);
SINKSAFE(I) := << I >> + MVDNSAFE(I) *
               ( */ <<SINKSAFE(J)|J IN PREDS(I) >>);
```

For hoisting, condition (1.b) must be altered as follows:

(b') IUSEVALSEQ(I,J) and IMODVALSEQ(I,J) must hold.

The following equations describe safety conditions for hoisting:

```
SAFE4(I) := <<J IN ND|IUSEVALSEQ(I,J) AND IMODVALSEQ(I,J)))>>;
MVUPSAFE(I)  :=  SAFE3(I) * SAFE4(I);
HSTSAFE(I)   :=  << I >> + MVUPSAFE(I) *
                 ( */ <<HSTSAFE(J)|J IN SUCCS(I)>>);
```

Turning our attention to profitability, we observe that for replication or sinking of a node J, the profitability tests are the same. Every node K on a path connecting J to the target must be examined to see whether its active set size is increased beyond MAXLIVE, by the variables used in the moving node.

In computing the new active set at node K, SET1, the set of variables used in J that are not active at K, must be added to the old active set, and SET2, the set of variables redefined in J must be removed from the old active set. The profitability test is expressed as follows:

$$|ACT(K) + SET1 - SET2| <= MAXLIVE$$

For hoisting of a node J, the profitability test is analogous at every intermediate node K. SET3, the set of variables modified in J that are not active at K must be

added, and SET4, the set of variables used in J that were active at K only because of the use at J must be removed. The profitability test is expressed as follows:

```
|ACT(K) + SET3 - SET4| <= MAXLIVE
```

For a practical implementation of this test, an incremental data flow analysis procedure which analyzes the effect of a small modification to the flow graph would be extremely desirable. In SOCRATES, SET2 and SET4 are approximated by <<Y>>, where Y is the variable being killed by the transformation. This results in an underestimate when other variables are also killed, and an overestimate when Y is not killed throughout the region, both of which may occur.

For an example of a profitable transformation, consider the following example reproduced from Section 2:

```
     DEF A(30),B(60),C(120),D(100),E(80),G(50),H(100);
1:   SET(B,A);
2:   SET(C) USE(A,B);
3:   SET(D) USE(A,C);
4:   SET(E) USE(A,B);
5:   USE(D,E);
6:   SET(H) USE(C,A);
7:   SET(G) USE(C,A);
8:   USE(G,H);
```

The safety sets are as follows:

```
SAFE1    = << 10001001, 11011001, 11110111, 11110111,
              11111111, 11111110, 11111110, 11111111 >>
SAFE2    = << 11010000, 11110110, 11110110, 11111110,
              11111110, 11111110, 11111111, 11111111 >>
SAFE3    = << 10001001, 01011001, 00110111, 01110111,
              11001111, 00111110, 00111110, 11111001 >>
SAFE4    = << 10000000, 11010000, 11110110, 11110110,
              11111110, 11111110, 11111110, 11111111 >>
```

```
INSTSAFE = << 10000000, 11010000, 1110110, 11110110,
             11111110, 11111110, 11111110,11111111 >>
MVUPSAFE = << 10000000, 01010000, 00110110, 01110110,
             11001110, 00111110, 00111110, 11111001>>

MVDNSAFE = << 10000000, 01010000, 00110110, 01110110,
             11001110, 00111110, 00111110, 11111001 >>
COPYSAFE = << 10000000, 11000000, 11100000, 11110000,
             11111000, 11111100, 11111110, 11111111 >>
SINKSAFE = << 10000000, 01000000, 00100000, 00110000,
             00001000, 00001100, 00001110, 00001001 >>
HSTSAFE  = << 10000000, 01010000, 00110110, 00010110,
             00001110, 00111110, 00111110, 11111111 >>
```

If conflict elimination between C and E is attempted, a downward move of statement 2 to statement 5 is profitable because only B's live extent is increased, while C's live extent is decreased, and B is smaller than C. The sink of statement 2 to statement 5 is not permissible, because C is used in statement 3, but the copy of statement 2 after statement 5 is permissible.

5.4 SOCRATES Implementation

In the SOCRATES procedure XFORM, the SAFEi vectors, i = 1,...4, are calculated directly for each node in the program, and the COPYSAFE, SINKSAFE and HSTSAFE sets are computed iteratively. Basic transformations are then attempted for each pair of variables in VSETMSK, the procedure parameter. For each such pair IV, IV1, the set of loops and nodes at which they conflict are examined to determine the type of conflict. If any type 3 conflict is found, the loops or nodes to be split are reported, and no further transformations are attempted. If no type 3 conflict is found, procedure BRKCONFLS is invoked twice to break the type 2 conflicts.

BRKCONFLS has three parameters: IX; IY; and TRYMSK, the set of nodes at which IY is live and IX is used or modified. The following SETL code describes BRKCONFLS:

```
BRKCONFLS: PROC(IX,IY,TRYMSK);
  HSTTARGS := NL;
    (FORALL I IN TRYMSK)
      AVAIL := <<N IN PAVIN(I)(2) |
                 (EXISTS IR IN MODS(N)|IY = VAR(IR))>>;
      OK := FALSE;
      TSTPROFDN(I, AVAIL, OK);
        (FORALL J IN AVAIL WHILE OK)
        OK := FALSE;
          IF J IN SINKSAFE(I)
            THEN OK := TRUE;
              CONTINUE FORALL J;;
            ENDIF;
          IF J IN COPYSAFE(I)
            THEN OK := TRUE;
            ENDIF;
        END FORALL J;
      IF NOT OK
        THEN
          IF AVAIL * HSTTARGS = NL
            THEN HSTTARGS := HSTTARGS + AVAIL;
              TRYHOIST(I,AVAIL,OK);
            ENDIF;
      IF OK
        THEN I FROM TRYMSK; ENDIF;
      END FORALL I;
    END BRKCONFLS;
TSTPROFDN: PROC(I,AVAIL,OK);
  OK := FALSE;
  (FORALL J IN ND | J > MIN(AVAIL) AND J <= I)
    IF IY IN ACT(J)
      THEN TESTVAL := MAXLIVE * |IY|;
      ELSE TESTVAL := MAXLIVE;
    NEWLIVE := <<+: DEREF(USES(N))|N IN AVAIL*PAVIN(J)(2)>>;
    IF |ACT(J) + NEWLIVE| > TESTVAL
      THEN RETURN;;
    END FORALL J;
  OK := TRUE;
  END TSTPROFDN;
```

```
TRYHOIST: PROC(I,AVAIL,OK);
  OK := FALSE;
    (FORALL J IN AVAIL)
      IF NOT I IN HSTSAFE(J) THEN RETURN;
      END FORALL J;
  NEWLIVE :=  <<+: DEREF(MODS(I))>>;
    (FORALL J IN ND|J > MIN(AVAIL) AND J <= I)
      IF IY IN ACT(J)
        THEN TESTVAL := MAXLIVE - |IY|;
        ELSE TESTVAL := MAXLIVE;
      IF |ACT(J) + NEWLIVE| > TESTVAL
        THEN RETURN;
      END FORALL J;
  OK := TRUE;
  END TRYHOIST;
```

6. OTHER TRANSFORMATIONS AND TECHNIQUES

A number of other techniques are applicable to this problem, although they were not explored in SOCRATES. With most of these techniques, the question of balancing the cost in execution time against the profit in execution space becomes critical. Some of these other techniques are discussed below:

6.1 Data Fragmentation

Data fragmentation is a technique that can enhance storage optimization. Often the storage layout produced by the overlay heuristic can be improved by selecting a particular variable to be fragmented and stored in several, nonadjacent storage segments. Since the live value analysis can determine subreferences that become dead at nodes where other subreferences of the same variable are still live, these may also serve as a guide for data fragmentation. Improvement in storage utilization may result. The object code that accesses this array must be cognizant of the number and extent of such fragments, and there is, therefore, a cost in object code execution time as well as code space.

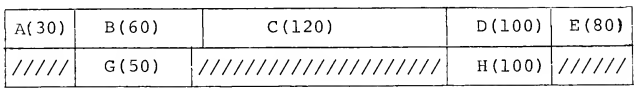
This is an interesting area for future research.

6.2 Data Spill

Data spill is an alternative to redundant code insertion. It is often preferable in register optimization, where the cost of register spill may not be prohibitive, but usually less desirable in storage optimization, where the cost of dumping a variable onto auxiliary storage between uses, and restoring into a renamed variable, may be very high. Consider the following example:

```
DEF A(30),B(60),C(120),D(100),E(80),G(50),H(100);
SET(C) USE(A,B);
SET(D) USE(A,C);
SET(E) USE(A,B);
USE(D,E);
SET(H) USE(C,A);
SET(G) USE(C,A);
USE(G,H);
```

The following storage layout is best for this program:

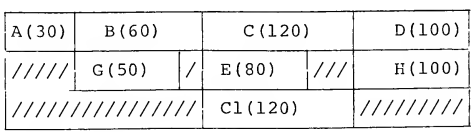


Total: 390

By saving and restoring C, and using a renamed variable for the result, we get the following program:

```
DEF A(30),B(60),C(120),C1(120),D(100),E(80),G(50),H(100);
SET(C) USE(A,B);
SET(D) USE(A,C);
USE(C);          /* this is the save */
SET(E) USE(A,B);
USE(D,E);
SET(C1);         /* this is the restore */
SET(H) USE(C1,A);
SET(G) USE(C1,A);
USE(G,H);
```

The following storage layout is best for this program:



Total: 310

This saves 80 words of object storage, at the expense of instruction storage and program execution time.

Data spill can be combined with redundant code insertion when several recomputable variables require a common input variable that can be spilled.

Cost considerations are critical for automation of the data spill technique.

6.3 Redundant Code Elimination

Consider the following example

```
DEF A(30),B(60),C(20),CC(20),D(10),DD(10),E(80),EE(80);
1: SET(C) USE(A,B);
   SET(D) USE(B,C);
   SET(E) USE(C,D);
   USE(D,E);
   SET(CC) USE(A,B);     /* this is a copy of 1 */
   SET(DD) USE(CC);
   SET(EE) USE(CC,DD);
   USE(DD,EE);
```

The following storage layout is best for this program:

A(30)	B(60)	C(20)	D(10)	E(80)
////////////////		CC(20)	DD(10)	EE(80)

Total: 200

If the common expression is identified, the following program results:

```
DEF A(30),B(60),C(20),CC(20),D(10),DD(10),E(80),EE(80);
SET(C) USE(A,B);
SET(D) USE(B,C);
SET(E) USE(C,D);
USE(D,E);
SET(DD) USE(C);
SET(EE) USE(C,DD);
USE(DD,EE);
```

The following storage layout is best for this program:

A(30)	B(60)	C(20)
D(10)	E(80)	//////
DD(10)	EE(80)	//////

Total: 110

This saves 90 words of object storage, as well as instruction storage and program execution time.

In order to implement this in a projected version of SOCRATES, the SOL language (see Section 7.3) would have to be extended to permit the specification of redundant code segments. In particular, the statement

EQUIV label-list

could be used to specify that the statements whose labels are listed are computationally equivalent.

6.4 Loop Fusion and Rank Reduction

Rank reduction can be performed when an array is dead on entry to a loop, and dead on exit from the loop, and when the subsets of the arrays that are referenced on each iteration of the loop are disjoint. The intent is to reduce the size of one of the large variables in the maximum clique set. Rank reduction can often be used in conjunction with loop fusion to reduce storage requirements. Consider the example of Section 2.3.6, which was:

```
        DEF A(100), B(100);
        SET(A,B);
        SET(SUMA,SUMB);
LP:     DO USE(I) SET(I) TEST;
           SET(SUMB) USE(SUMB,I,B);
           END LP;
        USE(SUMA,SUMB);
```

This was transformed by a loop splitting transformation into the following form:

```
        DEF A(100), B(100);
        SET(A);
        SET(SUMA);
LP:     DO USE(I) SET(I) TEST;
           SET(SUMA) USE(SUMA,I,A);
           END LP;
        USE(SUMA);
        SET(B);
LP1:    DO USE(I1) SET(I1) TEST;
           SET(SUMB) USE(SUMB,I1,B);
           END LP1;
        USE(SUMB);
```

In this example, a further transformation can be effected via rank reduction and loop fusion. If each (implicit) input loop is fused with the succeeding computation loop, the rank of each array can be reduced, resulting in the following program:

```
        SET(SUMA);
  LP:   DO USE(I) SET(I) TEST;
           SET(A);
           USE(SUMA) SET(SUMA,I,A);
           END LP;
        USE(SUMA);
        SET(SUMB);
LP1:    DO USE(I1) SET(I1) TEST;
           SET(B);
           USE(SUMB) SET(SUMB,I1,B);
           END LP1;
        USE(SUMB);
```

Only three words of data storage are needed now.

The specification of fusable loops also requires a SOL extension. The EQUIV statement can specify loop labels that delimit computationally similar loops.

6.5 Instruction Block Overlay

The automatic data overlay heuristic can be used to manage instruction storage as well as data storage. If the program is segmented into instruction blocks, then each block B can be treated like a variable, with its size specified. A common back dominator of the instructions in B should be chosen as a definition point for B, and all the instructions in B should be treated as use points for B.

In packaging the instruction blocks, and selecting appropriate definition points, the SOCRATES experimenter or the high-level language programmer using a storage optimizer must consider the execution cost of alternatives. This packaging of instructions is part of the work the programmer performs traditionally in defining a program overlay structure to a system overlay facility. The other part — determining block sizes and using them to derive an optimal sharing pattern — can be done by the storage optimizer.

The automation of instruction packaging using a cost function is an area for future research.

6.6 Inter-procedural Overlay

There are two ways in which interprocedural overlay can proceed:

(1) Macro overlay

The procedure is expanded in macro-like fashion when the flow graph FG is built. The procedure's local variables and instructions are treated like renamed variables at each call. A single conflict graph contains all the conflicts in the program, and is used by the overlay procedure.

(2) Piecewise overlay

Overlay is performed for each procedure in turn, in inverse call order. The procedure's local variables and instructions are overlaid, resulting in a collection of segments (the more small segments, the better), including

instruction storage. These segments are treated as "renamed variables" at each call to the procedure, and hence can be assigned locally optimal storage at each call.

SOCRATES can be easily extended to permit procedure definition in a PROC statement.

Again, the question of cost is relevant.

7. PROJECT DESCRIPTION: SOL AND SOCRATES

In order to provide empirical information about the storage optimizability of the average program, SOL, a Storage Optimization Language, has been designed, and SOCRATES, a Storage Optimization Code Reorganization And Transformation Experimental System, has been implemented.

The purpose of SOL is to express information about a program that is essential for automatic storage optimization. It is intended as an intermediate language that could serve as a common target for compilers of most languages (e.g., FORTRAN, COBOL, BASIC, PL/1, assembly language). SOL could be input either to an advice-giving program, such as SOCRATES, which examines and reports storage optimization possibilities, or to an actual storage optimizer.

Versions of the algorithms in Chapters 3-5 have been implemented in SOCRATES so that their applicability to actual programs can be assessed.

Section 7.1 contains an overview of SOCRATES, and SOL is specified in Section 7.2. The last section describes the program analysis phase of SOCRATES, and presents generalized redundancy equations for live/dead analysis and available expression analysis of arrays and array subreferences.

7.1 An Overview of SOCRATES

SOCRATES accepts the Storage Optimization Language as input that describes a program to be analyzed for its amenability to storage optimization. The SOL input is followed by a string of SOCRATES commands that indicate the analysis to be performed.

No transformations are actually performed on the SOL program. Instead, analyses are performed, as directed by the commands, and the results of these analyses are printed.

SOCRATES can be used in an iterative fashion: from the information printed for the initial SOL program, a new SOL program can be constructed by the experimenter, reflecting the suggested transformations, and SOCRATES can be rerun on this new program. A storage overlay heuristic may be invoked each time to monitor improvements. The purpose of SOCRATES is to provide empirical data for the ultimate design of a practical storage optimizer.

The user defines the flow of his program, and the way in which the data is used, defined and modified. The user can also describe certain data relationships which might be ascertained by range analysis. A special pair of statements is provided for the bracketing of single-entry, single-exit loops. The language is primitive enough to describe the essentials of any language, as long as static data extents are restricted to a constant. The program can be described statement by statement, basic block by basic block, or procedure by procedure, according to the experimenter's inclination.

Using these primitives, the experimenter can also describe blocks of instruction storage (see Section 7.5) and analyze a multiprocedure program (see Section 7.6). In the future, it is envisioned that front ends be built for several langauges, funneling into a common range analysis phase. Other possible SOCRATES extensions are discussed in Section 9.

7.2 The Storage Optimization Language

Although it is input to SOCRATES, the Storage Optimization Language has been designed as an intermediate-level language suitable for input to the storage-optimizing portion of a compiler. As such, one can assume that a front end process has parsed the program, gathered and analyzed control and data flow information, and performed a range analysis. This range analysis has translated subscripted array references

into symbolic subreferences, and determined the disjointness and covering properties of the subreferences. Thus SOCRATES or a hypothetical optimizer should be given enough information to deduce deadness and value availability of variables and subreferences to a refined degree (see Section 2.2).

For the purposes of this initial study, SOL programs were coded manually.

7.3.1 SOL Objects — Variables, References and Labels

The primitive objects in SOL are labels of SOL statements, variables which are to be arranged in storage, and references to those variables.

An identifier consists of one to eight alphanumeric characters. Identifiers are used for variables, subreferences and statement labels. There is no distinction between a program variable and a program constant in this language, so identifiers can begin with a digit.

A reference to a variable has one of the following forms:

(1) VAR

This is a reference to the entire variable VAR, where VAR is an identifer.

(2) VAR . SNAME

This is a reference to a subreference SNAME of VAR, where VAR and SNAME are identifiers.

SOCRATES will analyze the definition/use relationships of these references in order to produce refined information about variable activeness and value availability. This analysis is assisted by information povvided by the SOL user as to the disjointness of these subreferences and the covering relationships.

In performing storage assignment, an array will be stored in contiguous storage. That is, there is no attempt to improve overlay by storing disjoint array subreferences in nonadjacent storage blocks, although there are cases where storage could be reduced if this were attempted (see Section 6.1). Data fragment techniques are best addressed in a follow-on effort.

7.3.2 SOL Statements and SOCRATES Commands

SOCRATES accepts as input a batch of one or more storage optimization problems. Each storage optimization problem consists of a string of SOL statements, representing the program to be analyzed, followed by a string of SOCRATES commands, indicating the analysis to be performed. As in PL/1, every statement or command must be terminated by a semicolon (;), and may be preceded by one or more labels, each label terminated by a colon (:).

All statements, except the imperative statement, have a verb at the beginning of the statement. Statements contain one or more keyword options, whose paramters, when present, are enclosed in parentheses.

The following is a list of the SOL statements with brief descriptions:

Imperative Statement

The imperative statement describes the use and modification of data and the flow successors of a single node in the program. The granularity of this node (single statement, basic block, procedure) is at the user's discretion. Operational details are not specified, since they are irrelevant to the storage optimizer.

Definition Statement

The definition statement (DEF) is used to define the size of one or more variables, the disjointness of subreferences of these variables and/or the covering relationships among subreferences. It corresponds roughly to a source program declaration.

Loop-Delimiting Statements

The DO and END statements are used to delimit the start and end of a single-entry loop. Options on the statement describe the use and modification of data and whether loop exit testing is performed at the node.

Propagate Statement

The PROP statement describes the propagation (see section 2.4) of modifications into definitions at specified nodes.

These statements are specified in detail in Sections 7.3.3-7.3.6.

Each SOCRATES command consists of a single keyword, possibly followed by additional information, terminated by a semicolon. The commands are:

OVLAY n
The OVLAY command invokes overlay heuristic n (see Section 3.4).

RENAM var-spec
The RENAM command invokes the unsharing transformation which determines the "correct number of names" for each variable specified by var-spec (see Section 4.2.1).

REPOS var-spec
The REPOS command determines the applicability of the repositioning transformation to break conflicts between each pair of variables specified by var-spec (see Section 4.2.2).

XFORM var-spec
The XFORM command determines which nodes and loops can be split in order to break use-use, use-set, and set-set conflicts between pairs of variables in the set specified by var-spec (see Section 5.3); and which nodes and loops can be moved or copied in order to break live-use conflicts between pairs of these variables.

var-spec is one of:
- an empty list, meaning all the variables in MAXCLIQ.
- *, meaning all the variables in the program.
- var-list, specifying the list of variables to be considered.

XEQ

The XEQ command terminates a string of SOCRATES commands. It may be succeeded by another SOL program.

7.3.3 The Imperative Statement and the Data Flow Options

The imperative statement consists of a string of data flow options and/or the GOTO option, in any order. The GOTO option is written:

GOTO(label-list)

The label-list is a string of one or more label identifiers, separated by a comma, or by blanks and an optional comma. Each label identifier should correspond to the label of an imperative statement in the program. This option specifies all the nodes to which control may transfer after the given node. If the GOTO option is omitted, it is assumed that control flows only to the imperative statement that immediately succeeds the given node.

The data flow options are the USE option, the MOD option, and the SET option.

The USE option is written:

USE(ref-list)

Ref-list is a string of references, separated by a comma or by blanks and an optional comma. This option specifies the variables and/or subreferences that are used at this node.

The MOD option is written:

MOD(ref-list)

Ref-list is a string of references, separated by a comma or by blanks and an optional comma. This option specifies the variables and/or subreferences that are modified at this node.

The SET option is written:

SET(ref-list)

Ref-list is a string of references, separated by blanks or commas. This option specifies the variables and/or subreferences that are completely redefined at this node.

7.3.4 Definition Statement

The definition statement consists of the statement verb DEF, followed by one or more definition groups, separated by commas. Each definition group supplies information about one or more subject references. The following forms are permitted for a definition group:

```
var( size )
var( size ) relationships
var . subreference relationships
```

where size is an integer and relationships is one of:

```
DISJ( subreference-list )
COVER( subreference-list )
DISJ COVER( subreference-list )
```

These options specify, respectively, a set of mutually disjoint subreferences, a set of subreferences that cover the subject reference, and a set of mutually disjoint covering subreferences. Subreference-list is a list of subreference names, separated by a comma or by a blank and an optional comma.

7.3.5 Loop-Delimiting Statements

The DO and END statements are used to delimit a single-entry loop. The END statement may specify the label of the DO it is ending, in which case the termination of intermediate loops is implied. Both statements may contain data flow options. In addition, either or both statements may contain a TEST option, indicating that the test for loop exit is performed at the node. If a DO or an END statement contains the TEST option, then one of the successors of the bracket node is the node immediately following the loop's END statement; the other successor of the DO is its physically next node; the other successor of the END is the DO node. Thus, four types of loops can be described: no exit, top exit, bottom exit, and top-and-bottom exit.

The loop should be a single-entry loop, and there should be no exits within the loop other than those at the delimiting statements. No check is made to verify this condition, however.

The END statement also terminates a SOL program.

7.3.6 Propagation Statement

The PROP statement is used to indicate points in the program where references modified elsewhere in the program become completely redefined by virtue of a looping flow of control. The PROP statement is written:

PROP(ref-list) option(label-list)

where ref-list is a list of references, label-list is a list of labels, and option is one of: UP, DOWN or LOOP.

The UP option specifies that the modification should be propagated upward into a redefinition at the specified label(s) for the purpose of live/dead analysis. This implies that the specified subreferences are dead on entry to the node indicated by the label.

The DOWN option specifies that the modification should be propagated downward into a redefinition at the specified label(s) for the purpose of available value analysis. This implies that the specified subreferences have been assigned values that are available on exit from the node indicated by the label.

The LOOP option specifies a loop or list of loops for which both the UP and DOWN options apply — the UP option to the DO statement and the DOWN option to the statement following the END statement.

For example, a single-entry, top-exit loop in which an array is modified one element at a time might cause the upwards propagation of that modification into a definition at the DO statement and the downwards propagation of the modification into a definition at the statement after the END statement.

No check is made as to whether the specified references are actually modified in some node of the loop.

7.4 Program Analysis in SOCRATES

Input to the ANALYZE phase of SOCRATES from previous phases includes the following tables:

IPROG, the set of nodes in the program, sorted into depth-first post-order. Information for each node includes the following:

- USES, the set of references used in the nodes;
- DEFS, the set of references defined in the nodes;
- MODS, the set of references modified in the nodes;
- SUCCS, the set of successor nodes;
- PREDS, the set of predecessor nodes.

VARS, the set of variables used in the program. Among the information in VARS is the SIZE of the variable.

REFS, the set of references used in the program. The information for reference IR includes a pointer to the variable's VARS entry and the subreference name SNAME, as well as the following sets:

- CONJ, the set of references not disjoint with reference IR.
- SUBSETS, the set of references that are subreferences of reference IR.

COVTAB, the set of covers described in this program. A cover is a pair (IR,SET), where IR is a reference and SET is a set of subreferences that cover IR.

RDMAP, the set of reference definition or modification pairs. Each entry in this ordered list consists of a pair (IR,ID), where IR is a pointer to the reference that is defined in the node whose depth-first post-order number is ID. This table is used for available value analysis.

GENKIL, node-specific generation and kill vectors for bit iteration. The following sets are included for each node I:

CJUSES, the set of references that are used, or are conjoint with a reference used, at node I;
CJMODS, the set of references that are defined or modified, or are conjoint with a reference defined or modified, at node I;
SSDEFS, the set of references that are defined, or are a subset of a reference defined, at node I;
KILL, the set of rdefs (RDMAP entries) definitely killed by a definition at node I.

ANALYZE includes the following procedures:

FINDTOTV -- calculates the total unoptimized storage needed by the program;

BLDLVSET -- calculates the active sets for each node;

FINDMAXLV -- finds MAXLIVE, as well as the set of nodes whose size is MAXLIVE;

BLDCG -- builds the conflict graph;

FINDMAXCLQ -- calculates the set of cliques in the conflict graph, and computes MAXCLIQ, the size of the largest clique. This routine also finds MCSETU, the set of nodes in some clique in the conflict graph whose size is equal to MAXCLIQ.

BLDLMAPS -- computes the following sets for each variable IV.

LNACT, the set of nodes and loops at which IV is active;
LNUSES, the set of nodes and loops at which IV is used;
LNMODS, the set of nodes and loops at which IV is modified or defined.

BLDAVSETS -- computes the set of values possibly available at each node.

In order to simplify the SOCRATES implementation, iteration was used in BLDLVSETS and BLDAVSETS. Lattice properties and the depth-first post-ordering of FG assure rapid convergence.

In the following sections, we describe BLDLVSET, BLDCG, FINDMAXCLQ, and BLDAVSETS in greater detail.

7.4.1 BLDLVSET

BLDLVSET calculates the active sets for each node. Suppose SSDEFS, CJUSES and SUCCS are as described above. Let DEDSET(I) denote the set of references definitely dead at node I. Then the following pair of equations defines DEDSET:

```
DEDSET1(I) = * : << ( DEDSET(K) + SSDEFS(K) ) - CJUSES(K)
                        | K IN SUCCS(I) >>
 DEDSET(I) = DEDSET1(I)
           + << R IN REFS | ( EXISTS K <= NCOVS
                        | R = COVTAB(K)(1) AND
                      (FORALL J IN COVTAB(K)(2)
                        | J IN DEDSET1(I) ) ) >>
```

This equation is computed in an iterative manner. First, DEDSET1 is computed for all nodes until there is no change, then DEDSET is computed and the process is repeated until there is no further change in the DEDSETs.

7.4.2 BLDCG

BLDCG builds the conflict graph CG. The nodes of CG correspond to the variables in the program. For each IV in VARS, let ADJAC(IV) denote the adjacency set of variable IV. Then we have:

```
ADJAC(IV) := << IV1 IN VARS | ( EXISTS I IN ND | IV IN ACT(I)
                        AND IV1 IN ACT(I) ) >>;
```

This is equivalent to:

```
ADJAC(IV) := ( + / << ACT(I) IN LS | IV IN ACT(I)  >>)
                        -  <<IV>> ;
```

7.4.3 FINDMAXCLQ

FINDMAXCLQ calculates the set of cliques in the conflict graph, using the algorithm of Paull and Unger that is described in Ewen (Ew):

```
Suppose <<IV, IV=1..NVARS >>, are the nodes of CG.
Let I = 1, and let CS(1) = << << 1 >> >>.
Repeat the following steps until I > NVARS:
(i)   TEMP := NL;
(ii)  (FORALL S IN CS(I))
           TEMP := TEMP + ADJAC(IV+1)*S + <<IV+1>>;
(iii) TEMP1 := TEMP + CS(I);
(iv)  CS(I+1) := << S IN TEMP1
                      |(FORALL T IN TEMP1|
                      ((S = T)
                     OR (S+T NOT = T) ) ) >>
```

MAXCLIQ, the size of the largest clique, is found. This routine also finds MCSETU, the set of nodes in some clique in the conflict graph whose size is equal to MAXCLIQ.

7.4.4 BLDAVSETS

The BLDAVSETS procedure computes each node's PAVIN (and PAVOUT), the set of values possibly available on entry to (and on exit from) the node, by iterating forward through the depth-first post-order, repeatedly until there is no change in any PAVIN, computing the following bit equation:

```
PAVOUT(J) := GEN(J) + PAVIN(J) - DKILL(J)
PAVIN(I)  := +: << PAVOUT(J)|J IN PREDS(I)>>;
```

PAVIN(I) is the set of rdefs possibly available on entry to node I. PAVOUT(J) is the set of rdefs possibly available on exit from node J. Each rdef is an RDMAP entry, as defined at the beginning of Section 7.4.

GEN(J) is the set of rdefs generated at node J, and DKILL(J) is the set of rdefs that are definitely killed at node J, as defined at the beginning of Section 7.4.

The set of rdefs possibly available on entrance to node I is equal to the union over all the predecessors J of I of the set of rdefs generated at J union the set of rdefs possibly available on entrance to J and not definitely killed in J.

8. EXPERIMENTAL RESULTS

In order to provide an experimental evaluation of the amenability of the typical program to the storage optimization techniques described in this paper, SOCRATES has been verified against the examples in this work, and run against three moderate-sized production programs that were hand-translated into SOL. The sample runs can be found in Appendix II, and the test runs in Appendix III. In addition, an alternate interface to the SOCRATES overlay heuristics was built, and over 130 graphs were processed by these heuristics to develop experimental evidence of their effectiveness. In this chapter, we report on the results of these efforts, and evaluate the implications for future work.

Of course, these results are not statistically significant, but only indicative of what a future expanded study might discover. Such a future study would accept source language programs as input, eliminating the manual translation into SOL. Parser-generator(s) would translate FORTRAN, COBOL, PL/1, PASCAL or assembly language programs into an intermediate language suitable for input to a common range analysis phase, which would generate the SOL program for input to SOCRATES.

8.1 SOCRATES' Results

The three programs tested were called GRPCALL, AVERAGE and CALC2. The empirical overlay results using the descending figure of merit overlay algorithm, may be summarized as follows:

	GRPCALL	AVERAGE	CALC2
TOT_STO	303725	9057	848
MAXLIVE	297851	9055	825
MAXCLIQ	297851	9055	825
ROOM	297851	9055	825
% saved	1.93	.02	2.71

Thus, for all three programs, the heuristic produced an optimum storage layout, with reductions in storage utilization ranging from 0.02% to 2.71%. Since MAXLIVE = MAXCLIQ in each case, the renaming transformation was not applicable.

In all three programs, the deviations from a shipbuilding approximation were relatively few.

When code-modifying transformations were attempted on these programs, only splitting was applicable.

A word of interpretation may be appropriate. The smallest of these programs is the PL/1 program AVERAGE, consisting of a little more than one page of source listing. Since it is relatively short, its use of storage is localized, and, in effect, hand-optimized. For such a program, automatic storage overlay provides little improvement. A visual inspection of the program, however, indicates that considerable savings could have been effected by loop fusion, rank reduction and code reordering.

The FORTRAN program CALC2 is the largest program, and, as one might expect, the storage savings here are the greatest. Interestingly enough, all the overlaid variables are scalars, leading one to guess that FORTRAN DO variables are prime candidates for overlay. (In fact, it is with DO variables that the FORTRAN programmer most often uses "storage-less variables".)

The PL/1 program GRPCALL involves the summary of statistical information for a set of data whose extents may become quite large. As the program is written, all the data is kept in storage, and various totals and summaries are accumulated and reported. Visual inspection of the program suggests that storage-optimizing code transformations such as redundant code insertion, loop fusion and rank reduction would have produced considerably improved results by permitting the data to be read (and reread) from external storage.

There are classes of programs that are quite likely to profit from these optimization techniques, but were not available for SOCRATES' testing. One type is the large, multimodule program, written by several programmers and maintained and modified over a long period of time. Another type is the large, assembly language program for a small machine.

The results of this initial study indicate that further study is warranted.

8.2 Testing the Overlay Heuristics

The three heuristics discussed in Section 3.5 (descending availability, ascending extended degree and ascending figure of merit), as well as the Hoffman bounded heuristic, were run against 131 different graphs. The first chart below summarizes the number of optimum results each heuristic obtained. The second chart summarizes the number of results within 20% of optimum.

CHART 1. OPTIMUM TOTALS

Test Case	AVSUM	NODSUM	FIGMER	ROWSUM	TOTAL
CG5S	9	8	10	10	12
CG2C	3	3	3	3	3
CG5S2A	12	3	12	4	12
CG5S2B	5	4	6	7	12
CG5S2C	7	3	7	1	12
CG8S	12	9	12	12	12
CG8SA	10	7	9	9	12
HPCG7	12	9	12	12	12
HPCG8	7	6	7	10	11
RMCTRY	23	22	23	21	25
SHIPBLD	4	4	6	5	8
TOTAL	104	78	107	94	131
TOTAL %	80	60	82	72	

CHART 2. OPTIMAL TOTALS

Test Case	AVSUM	NODSUM	FIGMER	ROWSUM	TOTAL
CG5S	10	8	10	11	12
CG2C	3	3	3	3	3
CG5S2A	12	3	12	6	12
CG5S2B	11	7	11	12	12
CG5S2C	12	10	12	7	12
CG8S	12	9	12	12	12
CG8SA	11	9	11	12	12
HPCG7	12	12	12	12	12
HPCG8	11	11	11	11	11
RMCTRY	25	22	25	25	25
SHIPBLD	6	8	8	7	8
TOTAL	125	102	127	115	131
TOTAL %	95	78	97	88	

Thus, the figure of merit heuristic, or a mixed strategy heuristic involving ascending availability and descending extended degree, appear equally effective on the class of graphs tested. A great deal of further experimentation is needed before conclusive results can be claimed.

9. CONCLUSION AND FUTURE DIRECTIONS

In this paper, a delineation of the fundamental issues of automatic storage optimization has been initiated. A structure for a storage-optimizing compiler has been proposed, and original algorithms for overlay determination and storage-optimizing code transformation have been presented.

Automatic storage overlay has been formulated as an extended graph coloring problem. An exact overlay algorithm and a family of heuristic overlay algorithms have been introduced. A bounded approximation algorithm, due to A. Hoffman, has been presented.

A canonical renaming algorithm has been demonstrated that always succeeds in reducing MAXCLIQ to MAXLIVE, and can be used to break up other cliques in the conflict graph. Renaming introduces new variables into the program to assume some, but not all, of the critical live conflicts. Such a transformation usually improves overlay results.

Another approach to storage-optimizing code transformation is to eliminate the live conflicts between a particular pair of variables. A procedure to perform redundant code insertion and code motion for the elimination of particular conflicts has been given, together with safety redundancy equations and profitability tests. Other storage-optimizing code transformations, as well as data fragmentation, data spill, instruction block overlay and inter-producedural overlay, have been addressed in survey fashion.

Generalized redundancy equations for array data flow analysis have been presented and implemented in SOCRATES, a Storage Optimization, Code Reorganization and Transformation Experimental System. SOL, a Storage Optimization Language, has been designed and is used in SOCRATES to express data flow and control flow properties essential to storage optimization. Versions of the overlay heuristics, the renaming algorithm, and

the code transformations have been implemented in SOCRATES, and experiments involving actual programs have been conducted on a limited basis.

Preliminary results indicate that automatic overlay and storage-optimizing code transformation are applicable in many circumstances. This suggests that the SOCRATES effort should be extended so that meaningful experimental results can be attained. These results could determine design trade-offs in a storage-optimizing pilot compiler.

The results of this work are applicable even to register optimization. The renaming transformation permits the compiler to determine where "move-register" operations are necessary and beneficial. Many of the code transformations discussed in this paper, such as code motion, redundant code insertion and elimination and data spill, are potential register-optimizing code transformations.

Once the SOCRATES study is completed, a pilot storage-optimizing compiler should be built. Many of the problems that are open issues in this paper — particularly in the area of storage-optimizing code transformation — would be addressed in a pilot compiler effort. The SOCRATES study should yield information on the value of alternate design approaches, and should be extended to provide additional information, if needed, to aid the development of a pilot compiler.

There are long-range implications of this study. If a compiler can perform range analysis, it can compute execution-time estimates. One can envision, then, a compiler that optimizes execution time, subject to storage constraints, and/or storage, subject to execution time constraints. In such a compiler, the trade-offs among various transformations might be computed automatically.

The language implications of this study are noteworthy. The benefits of a storage-less variable have been discussed in the introduction. Bliss has demonstrated that structured

programming languages simplify the task of program analysis and storage-optimizing code transformation. It may be that a nonprocedural language, such as Dataflow (Ko76), can have advantages in the small computer environment where storage must be minimal at both compile time and execution time, because prorams can be constructed rather than being analyzed and then transformed. It would be interesting to investigate whether the task of program construction is, in fact, simpler and less demanding of computer resources, than the tasks of analysis and transformation.

A number of open problems remain:

(1) Finding a smaller upper bound on CHR(CG), possibly in the shipbuilding case.

(2) Disproving or proving the Garey-Johnson result for the shipbuilding problem.

(3) An improved exact overlay algorithm, together with an average-value analysis of the algorithm. It may well be that the average execution time is considerably less than exponential.

(4) Further work on overlay heuristics, including execution time analyses and/or experiments.

(5) A proof that the results of one of the overlay heuristics do not deteriorate when MAXCLIQ is reduced by renaming and/or by code modification. Alternatively, another heuristic for which the property can be demonstrated.

(6) A proof that register-minimizing code transformation is still NP-complete if redundant calculation is permitted.

(7) Extension of the code transformation work in this paper:
   (a) Investigation of solutions to the problem of compounding code-modifying transformations so that successive groups of code can be moved or redundantly inserted.
   (b) Investigation of conditions under which multi-source sinks can be performed.

(c) Incorporation of redundant code elimination techniques into the code modification procedure so that multi-source hoists can be performed.

(d) Exploration of the cascading transformation problem so that conditions on code motion and insertion can be relaxed.

(3) Extension of redundancy equations and SOCRATES procedures to include transformations in Chapter 6.

BIBLIOGRAPHY

(AhHU74) Aho, Hopcroft and Ullman
Design and Analysis of Computer Algorithms
Chapters 4, 5, Addison-Wesley, 1974.

(AhHU77) Aho, Hopcroft and Ullman
Principles of Compiler Design, Addison-Wesley, 1977.

(AhJU76) Aho, Johnson and Ullman
Code Generation for Expressions with Common Subexpressions, Proc. 3 ACM Symp. on Principles of
Programming Languages, January 1976, pp. 19-31.

(AhU70) Aho, A. V., and Ullman, J. D.
Transformations on Straight-Line Programs
Proc. 2 Annual ACM Symp. on Theory of Computing
May 1970, pp. 136-148.

(AhU73) Aho, A. V., and Ullman, J. D.
Theory of Parsing, Translation and Compiling
Vol. 2, Prentice-Hall, 1973.

(Al71) Allen, F. E.
A Basis for Program Optimization
Proc. IFIP Congress 1971, North-Holland, 1971, pp. 385-90.

(Al74) Allen, F. E., Interprocedural Data Flow Analysis, Proc.
IFIP Congress, 1974, North-Holland, 1974, pp. 398-492.

(AlC72a) Allen, F. E., and Cocke, J.
A Catalogue of Optimizing Transformations.
Design and Optimization of Compilers, R. Rustin, ed.
Prentice-Hall, 1972, pp. 1-30.

(AlC72b) Allen, F. E., and Cocke, J.
Graph-theoretic Constructs for Program Control Flow
Analysis. IBM Research Report RC 3923, July 1972.

(AlC76) Allen, F. E., and Cocke, J.
A Program Data Flow Analysis Procedure,
Communications of ACM, Vol. 19, No. 3, March 1976.

(B77) Barth, J. M., An Interprocedural Data Flow Analysis Algorithm, Proc. 4 Annual Conf. on Principles of Programming Languages, 1977, pp. 119-131.

(C70) Cocke, J., Global Common Subexpression Elimination, SIGPLAN Notices, Vol. 5, No. 7, 1970, pp. 20-24.

(C71) Cocke, J., On Certain Graph-Theoretic Properties of Programs, IBM Research Report RC-3391, 1971.

(CS70) Cocke, J., and Schwartz, J.
Programming Languages and Their Compilers
Courant Inst. Math. Sci., 1970, N. Y.

(DaR66) Dantzig and Reynolds
Optimal Assignment of Computer Storage by Chain Decomposition. Report No. ORC-66-6, U. of Cal, 1966

(De78) Dewar, R.B.K. The SETL Programming Language, New York Univ., Courant Inst. Math. Sci., 1978.

(EBA72) Earnest, Balke and Anderson
Analysis of Graphs by Ordering of Nodes
Jour. of ACM, Vol. 19, No. 1, Jan. 1972, pp. 23-42.

(Ew) Ewen, Shimon, Algorithmic Combinatorics

(GaJ76) Garey and Johnson
The Complexity of Near-Optimal Graph Coloring
Jour. of ACM, Vol. 23, No. 1, January 1976.

(Go79) Golumbic, M. C.
Algorithmic Graph Theory and Perfect Graphs (tentative title), Academic Press, (to be published).

(GrW76) Graham, S. L. and Wegman, M.
A Fast and Usually Linear Algorithm for Global Flow Analysis, Jour. of ACM, Vol. 23, No. 1, January 1976, pp. 172-202.

(Hara) Harary, F. Graph Theory, Chapter 12.

(Harr75) Harrison, W.
Compiler Analysis of the Value Ranges for Variables
IBM Research Report RC5544, July 1975.

(He77) Hecht, M.
Flow Analysis of Computer Programs, Elsevier North-Holland, Inc., 1977

(HeU72) Hecht, M. S. and Ullman, J. D.
Flow Graph Reducibility
SIAM Jour. of Computing, Vol. 1, No. 2, 1972, pp. 188-202.

(HeU74) Hecht, M. S., and Ullman, J. D.
Characterizations of Reducible Flow Graphs
Jour. of ACM, Vol. 21, No. 3, 1974, pp. 367-375.

(HeU75) Hecht, M. S., and Ullman, J. D.
A Simple Algorithm for Global Data Flow Analysis Problems, SIAM Jour. of Computing, Vol. 4, No. 4, December 1975, pp. 519-532.

(KaU76) Kam, J. B. and Ullman, J. D.
Global Data Flow Analysis and Iterative Algorithms
Jour. of ACM, Vol. 23, No. 1, Jan. 1976, pp. 158-171.

(Ken71) Kennedy, K.
A Global Flow Analysis Algorithm
International Jour. of Computer Math., Vol. 3, p. 5-15.

(Ken75) Kennedy, K., Node Listings Applied to Data Flow Analysis, Proc. 2 Annual Symp. on Principles of Programming Languages, 1975, pp. 10-21.

(Ken76) Kennedy, K.
A Comparison of Two Algorithms for Global Data Flow Analysis
SIAM Jour. Comput., Vol. 5, No. 1, pp. 158-180.

(KenZ77) Kennedy, K., and Zucconi, L., Applications of a Graph Grammar for Program Control Flow Analysis Proc. 4 Annual Symp. on Prin. of Program. Lang., 1977, pp. 72-85.

(Ker70) Kernighan, B. W.
Ph.D. Thesis, Princeton

(Ker71) Kernighan, B. W.
Optimal Sequential Partitions of Graphs
Jour. of ACM, Vol. 18, No. 1, Jan. 1971, 34-40.

(Ki73) Kildall, G. A.
A Unified Approach to Global Program Optimization
Proc. 1 Symp. on Prin. of Program. Lang., 1973, 194-206.

(Kn71) Knuth, D. E. An Empirical Study of FORTRAN Programs, Software Practice and Experience, Vol. 1, No. 12, 105-34.

(Ko76) Kosinski, P.
Dataflow
Proc. 3 Symp. on Princ. Program. Lang., 1976.

(Log78) Logrippo, L.
Renaming and Economy of Memory in Program Schemata,
Jour. ACM, Vol. 25, No. 1, Jan. 1978, pp. 10-22.

(Lov76) Loveman, D. B.
Program Improvement by Source to Source Transformation
Proc. 3 Symp. on Prin. of Program. Lang., 1976, 140-52.

(LowM69) Lowry, E. and Medlock, C.
Object Code Optimization
Comm. ACM, Vol. 12, No. 1, Jan. 1969, pp. 13-22.

(PS77) Paige, B. and Schwartz, J. T.
Reduction in Strength of High Level Operations
Proc. 4 Symp. on Princ. of Program. Lang., 1977, 58-71

(RLe77) Reif, J. H. and Lewis, H. R.
Symbolic Evaluation and the Global Value Graph
Proc. 4 Symp. on Prin. of Program. Lang., 1977, 104-18.

(Scha73) Schaefer, M.
A Mathematical Theory of Global Program Optimization,
Prentice-Hall, 1973.

(Schw75) Schwartz, J. T.
Automatic Data Structure Choice in a Language of
Very High Level, Proc. 2 Symp. on the Prin. of
Program. Lang., 1975, pp. 36-40.

(Se75) Sethi, R.,
Complete Register Allocation Problems
SIAM Jour. of Computing, Vol. 4, No. , 1975, pp. 226 ff.

(SzWi68) Szekeres, G. and Wilf, H. S.
An Inequality for the Chromatic Number of a Graph
Jour. Combin. Theory, Vol. 4, 1968, pp. 1-3.

(Ta72) Tarjan, R. E.
Depth-First Search and Linear Graph Algorithms
SIAM Jour. Comput., Vol. 1, No. 2, Sept. 1972, 146-60.

(Ta74) Tarjan, R. E.
Finding Dominators in Directed Graphs
SIAM Jour. of Comput., Vol. 3, No. 1, March 1974, 146-60.

(Te74) Tenenbaum, A.
Type Determination in a Language of Very High Level
Courant Inst. Math. Sci., Report NSO-3, NYU.

(U73) Ullman, J. D.
Fast Algorithms for the Elimination of Common Subexpressions
Acta Informatica, Vol. 2, No. 3, pp. 191-213.

(Wa78) Warren, H. S.
Static Main Storage Packing Problems
Acta Informatica, Vol. 9, pp. 355-376.

(WJWHG75) Wulf, Johnsson, Weinstock, Hobbs and Geschke
The Design of an Optimizing Compiler
American Elsevier Publ. Co., 1975.

(Y71) Yershov, A. E.
The ALPHA Automatic Programming System
Academic Press, 1971.