Technical Report CS78002-R

AN ANNOTATED BIBLIOGRAPHY ON OPTIMIZATION OF PROGRAMS DURING COMPILATION

by

J.A.N. Lee, Wm. Thacker and T.C. Harrison

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg VA 24061

1978 March 25

INTRODUCTION

This annotated bibliography covers a collection of articles assembled by the authors in a Graduate level course on advanced compilation techniques. Surprisingly there has been little advancement in techniques of optimization since the reports of the late 1950's and early 1960's, and it is believed that many techniques utilized by early compiler writers* have never been published formally. Consequently certain techniques may have been lost or are being rediscovered now.

A definition of the techniques of compiler code generation optimization are included in Frances E. Allen and John Cocke, A Catalogue of Optimizing Techniques, [3]** and include:

Procedure Integration,
Loop Transformation,
Common Subexpression Elimination,
Code Motion,
Constant Folding,
Dead Code Elimination,
Strength Reduction,
Linear Function Test Replacement,
Instruction Scheduling,
Register Allocation,
Storage Mapping,
Shadow Variables,
Anchor Pointing,
Special Case Code Optimization, and
Peephole (or window) Optimization.

They also include in their catalogue (sic) "Parsing Methods" but this technique is more applicable to the optimization of a compiler than the generation of optimal code.

Where possible, the keywords included in each annotation correspond to the set of titles in the Allen and Cocke catalogue.

---

* according to John Backus in an upcoming article on the development of FORTRAN I, II and III to be presented at the "History of Programming Languages Conference" in Los Angeles, June 1-3, 1978.

** Numbers in brackets [...] indicate the article number assigned in this bibliography.

LISTING OF PAPERS

1. Local Optimizations.....
   John T. Bagwell, Jr.
2. Control Flow Analysis.....
   Francis Allen
3. A Catalogue of Optimizing Transformations.....
   F.E. Allen and John Cocke
4. Data Flow Analysis in Software Reliability.....
   Lloyd D. Fosdick and Leon J. Osterweil
5. Register Assignment Algorithm for Generation of Highly Optimized Code.....
   J.C. Beatty
6. Compiler Construction for Digital Computers.....
   David Gries
7. The Anatomy of a Compiler - 2nd edition.....
   J.A.N. Lee
8. Design Characteristics of the WATFOR Compiler.....
   D.D. Cowan and J.W. Graham
9. Index Register Allocation.....
   L.P. Horwitz, R.N. Karp, R.E. Miller, and S. Winograd
10. A Comment on Index Register Allocation.....
    P. Luccio
11. Index Register Allocation in Straight Line Code and Simple Loops.....
    Ken Kennedy
12. Generation of Optimal Code for Expressions via Factorization.....
    Melvin A. Breuer
13. Optimal Code for Serial and Parallel Computation.....
    Richard J. Fateman
14. Global Common Subexpression Elimination.....
    John Cocke
15. Expression Optimization Using Unary Complement Operators.....
    Dennis J. Frailey

32. An Algorithm for Coding Efficient Arithmetic Operations.....
Robert J. Floyd

33. The Generation of Optimal Code for Arithmetic Expressions.....
Ravi Sethi and J.D. Ullman

34. Optimal Code Generation for Expression Trees.....
A.V. Aho and S.C. Johnson

35. The Generation of Optimal Code for Stack Machines....
J.L. Bruno and T. Lassagne

36. PROGRAMMING: An Introduction to Computer Languages and Techniques.....
Ward D. Maurer

37. Compiler Code Optimization.....
Thomas J. McCabe

38. A Fast and Usually Linear Algorithm for Global Flow Analysis.....
Susan Graham and Mark Wegman

39. Program Improvement by Source-to-Source Transformation.....
David Loveman

40. Global Data Flow Analysis and Iterative Algorithms...
John Kam and Jeffrey Ullman

41. On the Automatic Simplification of Computer Programs.....
J. Nievergelt

42. PUFFT - The Purdue University Fast Fortran Translator.....
Saul Rosen, Robert Spirgeon, and Joel Donnelly

43. Program Optimization.....
David Loveman and Ross Faneuf

44. Peephole Optimization.....
W.M. McKeeman

45. High Speed Compilation of Efficient Object Code.....
C.W. Gear

46. A Study of Replacement Algorithms for a Virtual-storage Machine.....
L.A. Belady

47. An Axiomatic Approach to Code Optimization for Expressions.....
J.C. Beatty

48. An Approach to Global Register Allocation.....
Richard Karl Johnsson

49. Complete Register Allocation Problems.....
Ravi Sethi

50. The Design of an Optimizing Compiler.....
Wm.A. Wulf, R.K. Johnsson, C.B. Weinstock, S.O. Hobbs

Article 1

John T. Bagwell, Jr.

Local Optimizations

"Proceedings of a Symposium on Compiler Optimization", ACM SIGPLAN Notices, July 1970, vol. 5, no. 7, pp. 52-65.

Local Optimization, Subscript Calculations, Safety

A one pass environment is defined for the class of local optimizations discussed. A case is made that these local optimizations are very inexpensive to implement and in some cases make compilation cheaper. Several machine independent optimizations are given (eg. changing (-1)**n to a check on if n is even or odd). Next, machine dependent optimizations are shown. In both of these sets of examples, only special cases of program code are handled.

The third section deals with the optimization of subscript calculation. This is done by calculating all of the constant part of the index ahead of time. A brief discussion on exponentiation optimization is the content of the next section.

The paper is concluded by a discussion on the important topic of "safety". This comes about due to some optimizations changing the implications of some code (eg. I/I cannot be replaced by 1 if there is a possibility that I=0).

Article 2

Francis Allen

Control Flow Analysis

"Proceedings of a Symposium on Compiler Optimization", ACM SIGPLAN Notices, July 1970, vol. 5, no. 7, pp. 1-19.

Data Flow Analysis, Graph Theory, Global Optimization

Control flow analysis is needed in order to do certain global optimizations. It answers such questions as "is this an inner loop?","can this expression be removed from this loop?". The approach of using directed graphs to represent the control flow is presented.

A good review of basic concepts and terms needed in a discussion on directed graphs is given. Procedures are described for finding certain relations needed for global analysis.

This paper does a good job on describing properties and algorithms for graphs needed for global analysis, but there is nothing on how to apply this tool in real life compilers.

Article 3

Frances E. Allen and John Cocke

A Catalogue of Optimizing Transformations

"Design and Optimization of Compilers", Courant Computer Science Symposium 5, Prentice-Hall, 1972, pp. 1-30.

Optimizations

A summary and categorization of current popular optimizations is given. The categories briefly discussed are: Procedure Integration, Loop Transformations, Common Subexpression Elimination, Code Motion, Constant Folding, Dead Code Elimination, Strength Reduction, Linear Function Test Replacement, Carrys, Instruction Scheduling, Parsing Methods, Register Allocation, Storage Mapping, Shadow Variables, Anchor Pointing, Special Case Code Generation, and Peephole Optimization.

For each of these categories a brief explanation is given. And, in most cases an example is shown to demonstrate what is being done for a particular transformation.

Article 4

Lloyd D. Fosdick and Leon J. Osterweil

Data Flow Analysis in Software Reliability

Computing Surveys, vol. 8, no. 3, Sept. 1976, pp. 305-30.

Data Flow Analysis, Code Optimization, Dead Code Elimination,

While this paper is not directed by the authors towards the process of program optimization, the techniques provided (in the form of several algorithms) are applicable to the processes of dead code elimination, redundant code recognition and anomaly identification. The paper is initially tutorial in nature, covering the basic tenets of data flow analysis on the basis of graph theory and previous work by F.E. Allen* and others (see references). The two major algorithms presented are those to determine, based on prior knowledge of the data flow graphs of the programs being investigated, the "live" variables and the "available" subexpressions. The authors claim to have tested the efficiency of these algorithms by having incorporated them into a real system known as DAVE.

* Allen, F.E., "program Optimization", in Annual Review in Autcmatic Programming, Pergamon Press, New York, 1969, pp. 239-307.

Article 5

J.C. Beatty

Register Assignment Algorithm for Generation of Highly Optimized Code

IBM J. Res. Develop., vol. 18, no. 1, Jan. 1974, pp. 20-39.

Register Allocation, Local Optimization, Global Optimization

The algorithm presented permits a high level of optimization at both local and global levels. The algorithm consists of essentially three different parts. Registers are first allocated locally (i.e. deciding what data should reside in a register). Then global allocation and assignment is performed followed last by local assignment of registers, thus giving local assignment a higher priority.

It was shown through development of a prototype compiler that program size could be reduced about 25 percent as compared to the IBM FORTRAN H compiler. It was stated that the time required for global assignment (approximately half of the compilation time) appeared to be linear in the number of references. The global assignment algorithm is not extremely difficult to understand, but implementation in an efficient manner requires complex code and data structures.

Article 6

David Gries

Compiler Construction for Digital Computers

John Wiley and Sons, Inc., New York, 1971

Register Allocation, Dead Code Elimination Factoring, Folding, Common Subexpressions, Code Optimization, Strength Reduction, Replacement.

This text contains a chapter on "Code Optimization" (pp. 375-411) which covers briefly the general topics listed in the keyword list above. Gries classifies the optimization techniques into two groups; those which are performed on the source program and those which are performed on the object program level. The former class of techniques includes:

(1) Folding: peforming operations whose operands are known at compile time;
(2) Elimination of redundant operations;
(3) Moving operations whose operands are invariant out of the range of loops; and
(4) Reducing the strength of certain operations within loops (mainly converting multiplication to addition).

Descriptions of basic blocks and regions are presented, along with suggestions and examples of their usefulness and implementations.

The author does not specially categorize the techniques which are applicable to the object program level, however he does mention (mainly in the "historical" section) such techniques as register allocation, test replacement, eliminating dead assignments and optimization for parallel processors.

Article 7

John A. N. Lee

The Anatomy of a Compiler - 2nd edition

D. Van Nostrand Company, New York, 1974

Common Subexpression Elimination, Loop Transformations, Folding

This text treats the problem of program optimization within a compiler in a general manner, giving a detailed algorithm only for the case of common subexpression elimination. The algorithm is based on recognizing the equivalence of triplets and is fundamentally similar in concept to the procedure outlined by Gries[6]. However, the author presents a technique for triplet recogntion based on the use of prime numbers which can substantialy reduce the pattern recognition operations necessary.

Article 8

D.D. Cowan and J.W. Graham

Design Characteristics of the WATFOR Compiler

Proceedings of a Symposium on Compiler Optimization, ACM SIGPLAN Notices, July 1970, vol. 5, no. 7, pp. 25-36.

Throughput Optimization, Compiler Implementation

A description of the implementation of the WATFOR compiler is given. Explained in the article are the structures used for the symbol tables and the methods used to make debugging of erroneous programs (i.e. telling the user the statement number last executed before an interrupt and telling the user when he tries to use an undefined variable in a computation).

Article 9

L. P. Horwitz, R. M. Karp, R. E. Miller and S. Winograd

Index Register Allocation

JACM, vol. 13, no. 1, Jan. 1966, pp. 43-61.

Index Register Allocation, Register Allocation

A procedure is given for index register allocation, which yields an optimal allocation for "straight line" code. The criterion for optimality is the number of loads and stores. the procedure takes into consideration whether a particular register has been modified and will be used again, thus requiring a store operation.

At first a graph(dag) is created which indicates all possible register assignments as needed for a particular program. The problem is thus reduced to finding the shortest path, given that each branch has a weight associated with it(i.e. whether a load or store or no modification is needed). Later, methods are given which allow certain subgraphs to be eliminated, thus decreasing the size of the problem.

Article 10

F. Luccio

A Comment on Index Register Allocation

CACM, vol. 10, no. 9, Sept. 1967, pp. 572-4.

Index Register Allocation, Register Allocation

This short paper presents a procedure which decreases the enumeration required for optimal index register allocation by the procedure presented by Horwitz et. al. [9].

Article 11

Ken Kennedy

Index Register Allocation in Straight Line Code and Simple Loops

Design and Optimization of Compilers, Courant Computer Symposium 5, Prentice-Hall, 1972, pp. 51-64.

Index Register Allocation, Register Allocation

This short paper presents an extension of Horwitz et. al. [9], which decreases the enumeration for optimization of index register allocation and allows for optimization of index register allocation in simple loops as well.

Article 12

Melvin A. Breuer

Generation of Optimal Code for Expressions via Factorization

CACM, vol. 18, no. 6, June 1969, pp. 333-40.

Factoring, Common Subexpression Elimination, Parallel Processors

This paper presents an intriguing method for the recognition of common subexpressions in a set of linear equations. The proposed system is based on the ability to produce a canonical ordering for expressions which include only commutative operators, though extensions are given to overcome this restriction. In particular, this method solves the problem of recognizing subexpressions which are often camouflaged by the hierarchy of operators. For example, this method will pick out the common factor between the two expressions:

A + B + C
and A + D + C

whereas methods based on triplets (see Gries and Lee) fail in this aspect. However, this method is likely to be most effective where there are a large number of linear equations in a straight line sequence of code. Additionally, the author sets forth an assignment algorithm to minimize the number of temporary storage cells required for evaluating the derived factors.

Globally optimal results are not necessarily produced by this heuristic system; (see Fateman [13]) however, the author believes that the benefits are worth the additional effort.

Article 13

Richard J. Fateman

Optimal Code for Serial and Parallel Computation

CACM, vol. 12, no. 12, Dec. 1969, pp. 694-5.

Common Subexpression Elimination, Parallel Processors

This short communication documents a number of cases where the algorithm by Breuer [12] produces code which is non-optimal. The two specific cases are an expression which Breuer reduces to four multiplications and four additions, which Fateman shows can be completed with only three multiplications and three additions, and an expression whose result is highly dependent on the order of evaluation.

Article 14

John Cocke

Global Common Subexpression Elimination

"Proceedings of a Symposium on Compiler Optimization", ACM SIGPLAN Notices, July 1970, vol. 5, no. 7, pp. 67-85.

Common Subexpression Elimination, Data Flow Analysis

A boolean procedure which utilizes a special type of Gaussian elimination to find redundant computations is presented. The author shows how this representation has special properties and gives helpful suggestions on speeding up the Gaussian elimination step. The relationship between this boolean representation and directed graphs helps the reader better understand what is being described. Finally, a procedure is given to tell what is to be done with the reduced bit string.

Article 15

Dennis J. Frailey

Expression Optimization Using Unary Complement Operators

"Proceedings of a Symposium on Compiler Optimization", ACM SIGPLAN Notices, July 1970, vol. 5, no. 7, pp. 67-85.

Common Subexpression Elimination, Special Case Code Generation, Folding

A notation for describing code and defining algorithms utilizing the notion of "complement operators" being carried along with the operands is presented. Also described is a cononical form for expressions. Algorithms for detecting common subexpresions, and cancelling operands to constants (special case code generation) are the next topics of discussion. The relationship of the algorithms and the flow of data between them is given to aid in the description of implementation.

Article 16

Edward Lowery and C.W. Medlock

Object Code Optimization

CACM, vol. 12, no. 1, Jan. 1969, pp. 13-22.

Data Flow Analysis, Common Subexpression Elimination, Register Allocation, Code Motion

A discussion on the implementation of object code optimizations and possible extents utilized by the OS/360 FORTRAN H compiler is the main topic. Presented is how the "dominance" relationships are found which facilitate the elimination of common expressions across the whole program and identifies loop structures. Optimizations on the induction variable in loops is described. A novel boolean procedure is presented for aid in register allocation that promotes inner loops being assigned the most registers. Several optimization techniques not used by FORTRAN H are presented.

Article 17

W.H.E. Day

Compiler Assignment of Data Items to Registers

IBM Systems Journal, vol. 9, no. 4, 1970, pp. 281-317.

Register Allocation

This paper formulates as integer programming problems for assigning data items to registers in the compilation process -- the one-one, many-one, and many-few global assignment methods. Correspondingly, three algorithms are described for obtaining feasible solutions to the three methods, one of which provides an optimal solution. The others are shown to be good approximations (within 10% of the optimal) and are sufficiently fast for inclusion in production compilers.

The algorithms for these data-to-register assignments are given in the APL language, but were not implemented in this same language for testing.

Article 18

Robert W. Witty

The Switching Reverse Polish Algorithm

ACM SIGPLAN Notices, vol. 12, no. 9, Sept. 1977, pp. 114-23.

Register Allocation

This algorithm for the improvement of the evaluation order of expressions written in "Reverse Polish" notation, is based on the works of Nakata [30] and Redziejowski [28]. Consequently, the work reported is equally applicable to the optimization of the allocation of registers to the evaluation sequence of the original expressions. In this paper, optimization is taken to mean the minimization of the size of the execution-time stack needed to retain the operands and intermediate results during the evaluation process. Obviously, this stack size is directly analogous to the number of registers needed in a register machine.

The technique of optimization is based on the switching between two generation algorithms - a late operator form (producing such strings as ABC++ from A+B+C) and an early operator generator (producing AB+C+ from the same string).

The author claims the method to be fast, simple and effective. However, there are unsubstantiated claims* that the algorithm is incorrect.

* According to students in the Fall 1977 Honors section of CS 2071.

Article 19

A. V. Aho, S. C. Johnson, and J. D. Ullman

Code Generation for Expressions with Common Subexpressions

JACM, vol. 24, no. 1, Jan. 1977, pp. 146-60.

Common Subexpression Elimination

The problem of producing optimal code from programs with common subexpressions being NP-complete is addressed. Proofs are given to show that generating optimal code even for some subclasses of programs with common subexpressions is NP-complete. A nice description of why optimal code generation is difficult in terms of time follows. Due to this difficulty several heuristic techniques are explored which produce "nearer optimal" solutions. Also, analyses of these schemes are shown and worst case verses optimal ratios are given for each algorithm, along with a description of each worst case.

Article 20

A. P. Yershov

ALPHA - An Automatic Programming System of High Efficiency

JACM, vol. 13, no. 1, Jan. 1966, pp. 17-24.

Loop Transformations, Strength Reduction, Code Motion, Common Subexpression Elimination, Storage Minimization

The environment of the ALPHA compiler is described where one would note that there are only 4096 words of main memory. The optimizations noted in the keywords are performed over the code, along with a novel one called "global memory economy". The compiler produces an inconsistency matrix which determines when variables may not share the same memory cell. This allows near optimal allocation of main memory. Another interesting fact is that the compiler is divided into 24 blocks. At the end, statistics are given for time and space of programs translated by ALPHA and those written by "hand".

Article 21

F. E. Allen and J. Cocke

A Program Data Flow Analysis Procedure

CACM, vol. 19, no. 3, March 1976, pp. 137-47.

Global Optimization, Data Flow Analysis

A review of Allen's [2] paper is given along with defining a few new terms. The major contribution of this paper is not just algorithms to accomplish what they describe, but actual PL/1 programs. A more detailed description of the programs themselves would be more helpful in understanding what is being done. The paper concludes with some time and space bounds on their programs.

Article 22

Vincent A. Busam and Donald E. Englund

Optimization of Expressions in Fortran

CACM, vol. 12, no. 12, Dec. 1969, pp. 666-74.

Loop Transformations, Common Subexpression Elimination, Register Allocation

A three pass optimizing FORTRAN compiler produced by Computer Sciences Corporation is described in a general enough fashion that the ideas used can be applied to any type of compiler.

The first pass builds five tables needed for future passes, symbol dictionary, statement number dictionary, constant dictionary, expression dictionary and encoded source program. It is in the encoded source program that information about the flow structure of the program is contained.

Pass 2 is where actual optimizations are done, common subexpressions are eliminated and invariants are moved out of loops. This is done using a technique which utilizes "forward definition points" and "backward definition points" for variables and expressions. Pass 2 also determines register assignment for DO loops and optimizes subscript calculation.

Pass 3 is the code generator, which uses the information gathered from the previous 2 passes to generate optimized code.

Article 23

Christopher Earnest

Some Topics in Code Optimization

JACM, vol. 21, no. 1, Jan. 1974, pp. 76-102.

Common Subexpression Elimination, Code Motion, Dead Code Elimination, Safety

A review of the basic material presented by John Cocke [14] is presented as background. The use of bit string representation and logical operations is used for supposedly more efficient determination of redundant expressions. Extensions from Cocke's work are given that determine dead code. A more extensive discussion of safety and profitability along with an evaluation of the algorithms presented conludes the paper.

Article 24

Donald E. Knuth

RUNCIBLE - Algebraic Translation on a Limited Computer

CACM, vol. 2, no. 9, Sept. 1959, pp. 18-21.

Folding

This paper is referenced in several other papers on optimization techniques and is included here for completeness. This article presents the "algorithm" for the compilation of arithmetic expressions using an IBM 650 computer with added alphabetic devices. The algorithm is presented in terms of a complex flowchart (definitely not structured) and which is in terms of a set of machine operations. The algorithm contains some degree of optimization, in that, in the right to left scan of the expression "worst case" instructions are developed which are later eliminated. This process is referred to in this paper as "decompiling."

Article 25

Harry D. Huskey

Compiling Techniques for Algebraic Expressions

Computer Journal, vol. 4, Jan. 1961, pp. 10-19.

Optimizations

This paper describes a method of translating algebraic formulae into a computer program in a single pass. Although not a paper explicitly dealing with optimization techniques this paper is included here for completeness since it is referenced by several other papers. The method presented is a predecessor of operator precedence grammars. The only explicit reference to optimization is the statement "Some of these (generated instructions) are redundant and could be eliminated with some complication in the logic."

Article 26

Billy C. Claybrook

A New Approach to the Symbolic Factorization of Multivariate Polynomials

Artificial Intelligence, vol. 7, 1976, pp. 203-41.

Factoring

A heuristic factoring scheme is presented which is shcwn to be a considerable improvement over previous factorization techniques and which is able to learn from previous experiences. While the technique presented in the paper is not explicitly directed towards the optimization of code, its applicability to the recognition of common (sub)expressions is obvious. In particular, the method determines the set of "unsimplifiable" factors which would be useful in the task of common (sub)expression elimination. However, the method may be a case of "overkill" for most commcnly found expressions in programs and has not been extended to the case of identifying factors in simultaneous multivariate polynomials.

Article 27

James E. Anderson

A Note on Some Compiling Algorithms

CACM, vol. 7, no. 3, March 1964, pp. 149-50.

Arithmetic Expressions, Stack Machines

This paper is one of the first papers written about register allocation optimization. It is suggested that code generation for expressions can be deferred until the entire expression has been examined, and the operator hierarchy be preserved by recording the structure of the expression in a tree. If the tree is properly manipulated, redundant stores and loads may be eliminated.

A routine is given for producing an expression tree. And an algorithm is given for traversing the tree and producing code for a machine with a single accumulator (a flowchart of this algorithm is included). The algorithm maintains the order of operation as specified in the expression.

Article 28

R. R. Redziejowski

On Arithmetic Expressions and Trees

CACM, vol. 12, no. 2, Feb. 1969, pp. 81-4.

Arithmetic Expressions, Graph Theory, Storage Minimization

An algorithm is described which allows an expression tree to be drawn in such a way that the number of accumulators needed for the computation can be represented in a straight-forward manner. The choice of best order of computation is thus reduced to the specific problem of a "minimum width" ordering of an expression tree.

A proof of the algorithm is given which also serves as a proof for the algorithm presented by Nakata[30].

Article 29

R. A. Freiburghouse

Register Allocation Via Usage Counts

CACM, vol. 17, no. 11, Nov. 1974, pp. 638-42.

Common Subexpression Elimination, Register Allocation

A method of register allocation is presented which makes use of the number of times that a value is used. The problems of value retention and register demand are looked at carefully. By testing many different program types it was found that generally fewer loads are generated using usage counts than either the least-recently-used criterion or the least-recently-loaded criterion.

Article 30

Ikuo Nakata

On Compiling Algorithms for Arithmetic Expressions

CACM, vol. 10, no. 8, August 1967, pp. 492-4.

Arithmetic Expression, Register Allocation

An algorithm is presnted which is an extension of the algorithm presented by Anderson [27], by including cases where there are N accumulators. The only additional information which must be included in the creation of the expression tree is an indicator of the number of accumulators necessary for calculating the expression corresponding to each particular branch without a store instruction.

A method is given for deciding the number of accumulators needed to calculate an expression a*b, where a and b are expressions and * is one of the operators +, -, *, and /. An algorithm is given in program form (PASCAL-like) for generating object code from a tree given N accumulators.

Article 31

John Bruno and Ravi Sethi

Code Generation for a One-Register Machine

JACM, vol. 23, no. 3, July 1976, pp. 502-10.

Register Allocation, Arithmetic Expressions, Code Optimization

This paper presents a formal proof that the generation of minimal-length code for machines that use accumulators or registers is NP-complete. The problem is first shown to be NP-complete for a machine which has a single register. It is then hypothesized that code generation for multiple-register machines is also NP-complete.

Article 32

Robert W. Floyd

An Algorithm for Coding Efficient Arithmetic Operations

CACM, vol.4, no.1, Jan. 1961, pp. 42-51.

Common Subexpression Elimination

This early article on code generation suggests the use of two techniques to improve code generation. The first is the observation that right-to-left scanning of an arithmetic expression will often (for single accumulator machines) generate more efficient code than left-to-right scans. Although not explicitly reasoned, it is assumed that Floyd recognizes the problems involved in stacking operations in an arithmetic scan and the problem of retrieving left-most operands after the left-to-right scan has pushed the right hand operand on top. He points out that in certain cases, such as the scanning of subscripted variables the scanning direction needs to be modified.

The second algorithm presented is for the manipulation of subexpressions which include the unary negation operation or the use of the unary negation to develop common subexpressions.

Article 33

Ravi Sethi and J. D. Ullman

The Generation of Optimal Code for Arithmetic Expressions

JACM, vol. 17, no. 4, Oct. 1970, pp. 715-28.

Arithmetic Expressions, Code Optimization, Register Allocation, Commutativity, Associativity

Three algorithms with formal proofs are given which extend the work of Floyd[32], Anderson[27], Meyers* and Nakata[30]. The algorithms minimize the number of instructions needed to evaluate an arithmetic expression as well as minimizing the number of storage references needed to evaluate an expression using N registers. The algorithms yield the mentioned results acording to different assumptions concerning the operators of the expressions being evaluated.

The three algorithms each start with a binary expression tree similar to the ones proposed by Nakata and Meyers, but the nodes are labeled differently to handle noncommutativity of operators correctly.

Algorithm One assumes that all operators are noncommutative. This algorithm is essentially a top-down traversal of the binary tree created from the expression to be evaluated.

Algorithm Two assumes that certain operators are commutative. This algorithm requires a slight modification of the binary tree in cases where the left subtree of a node is a leaf and the right subtree is not a leaf (flip the subtrees). After modifying the tree, traversal is the same as in Algorithm One.

Algorithm Three assumes that certain operators are both commutative and associative. After creating a binary tree, the tree is reordered into an associative tree, which may require much manipulation of the structure of the tree. The nodes are then labeled according to labeling rules for an associative tree. Then during traversal (code generation) the descendants of associative nodes must be reordered, and then proceed as in Algorithm Two.

The three algorithms all run in linear time.

* Meyers, W.J., Optimization of Computer Code. Unpublished memorandum, G.E. Research Center, Schenectady, N.Y., 1965, (12pp).

Article 34

A. V. Aho and S. C. Johnson

Optimal Code Generation for Expression Trees

JACM, vol. 23, no. 3, July 1976, pp. 488-501.

Code Optimization, Arithmetic Expressions

A dynamic programming algorithm is presented which produces a program of minimal length for a broad class of machines. There is no common subexpression elimination (i.e. only expression trees are considered, no dags). In the conclusions of the article it is stated that the algorithm could be modified to use execution time or number of stores as its criterion for optimality.

The class of machines for which this algorithm works has a fixed number of general registers and a countable sequence of memory locations. It has simple load and store instructions. It also has operations which leave the result in a register. If registers are used by an operation, the result is stored in one of the registers used by the operation.

Article 35

J. L. Bruno and T. Lassagne

The Generation of Optimal Code for Stack Machines

JACM, vol. 22, no. 3, July 1975, pp. 382-96.

Code Optimization, Stack Machines

Three algorithms are presented for evaluating arithmetic expressions on a machine with a fixed number of registers configured as a stack. The three algorithms have the same restrictions as those in Sethi and Ullman [33]. The algorithms manipulate a binary expression tree in the same manner as Sethi and Ullman describe and then are traversed in such a manner that they yield optimal code in terms of transfer of information between the stack and main memory.

Article 36

Ward D. Maurer

PROGRAMMING: an Introduction to Computer Languages and Techniques

Holden-Day,Inc., San Francisco, 1968, pp. 199-203.

Register Allocation, Common Subexpression Elimination

This paper is referenced by Gries [6] as a source of data on compiler optimization. However, Maurer merely describes two optimization problems (register allocation and common subexpression recognition) without indications of the techniques of implementing the optimization processes.

Article 37

Thomas J. McCabe

Compiler Code Code Optimization

NSA Technical Journal, Special Computer and Information Sciences Issue, (undated).

Common Subexpression Elimination, Loop Transformations, Strength Reduction

This article is a survey of the "process of compiler optimization in terms of some of the specific techniques that optimizing compilers use and introduces some new techniques that have not been used in currently available compilers." Although the paper discusses several optimization operations, it does not (nor claims to) actually describe the optimization algorithms which are necessary to accomplish these optimization goals. As far as can be determined the only "new" technique which is introduced is "subsumption" which is the postponement of the formation of an array until its first point of usage. This article contains no references to any other work on optimization.

Article 38

Susan Graham and Mark Wegman

A Fast and Usually Linear Algorithm for Global Flow Analysis

JACM, vol. 23, no. 1, Jan. 1976, pp. 172-202.

Data Flow Analysis, Common Subexpression Elimination

The authors present a new algorithm for flow anlaysis. They show that this algorithm takes on a time bound proportional to the number of edges plus the number of exits frcm program loops. So structured programs that have one entry, one exit structure will only take linear time for the analysis. A very formal analysis with a fair amount of backgrcund definitions needed is given. A section on implementation helps put the algorithm and its applications in a practical perspective.

Article 39

David Loveman

Program Improvement by Source-to-Source Transformation

JACM, vol. 24, no. 1, Jan. 1977, pp. 121-45.

Code Motion, Loop Transformations

A series of optimizations that can be done at the source level (constant computation, expression simplification, loop collapsing, pruning, reordering conditionals, assignment elimination, common subexpression elimination, code motion, strength reduction, dead variable elimination, subsumption, GO TO chasing) are just mentioned and examples are given. Most of the paper is taken up with loop transforms since these are what take up most of a prccesses time. Good examples of each case are presented. The cnly problem is that nothing is mentioned about implementation and the worthiness of implementing an automatic process for these transformations. It is excellent, however, in showing a programmer better ways of coding.

Article 40

John Kam and Jeffrey Ullman

Global Data Flow Analysis and Iterative Algorithms

JACM, vol. 23, no. 1, Jan. 1976, pp. 158-71.

Data Flow Analysis, Common Subexpression Elimination, Folding

The authors present algorithms and theorems for code optimization through data flow analysis using a lattice theoretic framework. One structure relied upon heavily in this work is a depth-first spanning tree. An algorithm to create a DFST from a flow graph is given. Lattice theory results are applied to this structure. In presenting this, the authors made an assumption of some background knowledge by the reader. Time bounds are derived for finding data propagation and improved if more is known about the DFST. Reference is made frequently to Kildall's paper and is the main basis for this paper. Reading Kildall's paper first is almost essential.

Article 41

J. Nievergelt

On the Automatic Simplification of Computer Programs

CACM, vol. 8, no. 6, June 1965, pp. 366-70.

Code Motion, Common Subexpression Elimination

A description of a few machine-independent simplifications is presented. Some of the terminology used is not descriptive and not used today, but everything is well defined. When formalisms are required, the author keeps them as simple and understandable as possible. Questions that were raised in the conclusion include whether the situations which can be simplified occur in practice and if the effort to optimize is worthwhile. This is an excellent first paper for the study of optimization.

Article 42

Saul Rosen, Robert Spitgeon and Joel Donnelly

PUFFT - The Purdue University Fast Fortran Translator

CACM, vol. 8, no. 11, Nov. 1965, pp. 661-6.

Throughput Optimization

This paper, which is referenced by many others in this report, does not talk about optimizing techniques in compilers. Instead, the authors describe an environment where a large amount of processing is done for small student jobs. The point they wish to make is that in an environment such as theirs, an optimizing compiler would be very uneconomical. Rather than an optimizing compiler, they have a very simple FORTRAN compiler that does virtually no optimization. They conclude with statistics showing for their type of jobstream, the PUFFT system works much better than a system that does some optimization.

Article 43

David Loveman and Ross Faneuf

Program Optimization - Theory and Practice

ACM SIGPLAN Notices, vol. 10, no. 3, March 1975, pp. 97-102.

Folding, Common Subexpression Elimination, Strength Reduction, Code Motion

Presented is a description of the optimizations and how they are done in the ILLIAC FORTRAN compiler. An outline of local and global information needed to perform these optimizations is given. One thing needed, that is described further is a p-graph which tells where a variable gets a new value and shows the flow of the program. Next comes a brief description of each optimization done (Linearize array references, constant propogation, dead code elimination, scalar propogation, invariant code motion, strength reduction, test replacement, folding, common subexpression elimination). An algorithm is given that tells us how to create a p-graph. This algorithm has no restrictions on the program like some others.

Article 44

W.M. McKeeman

Peephole Optimization

CACM, vol. 8, no. 7, July 1965, pp. 443-4.

Code Optimization

The author gives a short description of how object code can be improved with a slight amount of effort. A local amount of code is looked at rather than the entire program. The level of complexity will depend on how much code is viewed at once. The techniques presented are easily applied and have been done at Stanford.

Article 45

C.W. Gear

High Speed Compilation of Efficient Object Code

CACM, vol. 8, no. 8, August 1965, pp. 483-8.

Common Subexpression Elimination, Code Motion, Folding

A three pass compiler is described where the last two passes scan an intermediate language. Pass 1 translates source input to a polish postfix notation. Pass 2 scans in reverse order and locates local common subexpressions. A unique method for accomplishing this with various stacks is described. Not much is said about pass 3 since that is where code is generated and is very machine-dependent. Overall, this paper is very nice in that everything said is practical and relatively easy to implement.

Article 46

L.A. Belady

A Study of Replacement Algorithms for a Virtual-storage Computer

IBM Systems Journal, vol. 5, no. 2, April 1966, pp. 78-101.

Virtual Storage, Page Replacement

This work is referenced by several of the articles in this bibliography. Within this article is an optimal replacement algorithm the ideas of which are used in several optimal register allocation algorithms. The most important idea is the "look-ahead" to see which pages (registers) are needed next and how soon they are needed.

Article 47

J.C. Beatty

An Axiomatic Approach to Code Optimization for Expressions

JACM, vol. 19, no. 4, Oct. 1972, pp. 613-40.

Arithmetic Expressions, Associativity, Commutativity, Code Optimization, Parallel Processors, Semantics

Two algorithms are presented which find optimal equivalent forms of an expression with no multiple references to any variable. The first algorithm is for parallel computers. The second algorithm extends the results of Sethi and Ullman[33] to include the unary minus. Optimality in this article is minimum code length.

Richard Karl Johnsson

An Approach to Global Register Allocation

Carnegie-Mellon University thesis, Dec. 1975

Register Allocation, Storage Minimization

In his introduction Johnsson gives a brief summary of the work done by Horwitz et. al.[9], Luccio[10], Nakata[30], Redziejowski[28], Sethi and Ullman[33], Beatty[5], Bruno and Sethi[31], and W.H.E. Day[17].

The article covers the allocation of registers and other temporary storage using the Bliss/11 compiler as an example. An attempt is made to formalize this phase of compilation and present some material to assist in development of a general model of the temporary storage problem.

The model described assigns a cost to each entity to be stored and each class of storage location, which helps in deciding the least expensive code for a program, but does not limit the decision to only two kinds of storage.

This paper has many ideas that were introduced in Wulf et. al.[50].

Article 49

Ravi Sethi

Complete Register Allocation Problems

SIAM J. Comput., vol. 4, no. 3, Sept. 1975, pp. 226-48.

Register Allocation

The objective observed in this paper was to limit the number of registers used in a straight line section of code (i.e. a sequence of assignment statements), while not permitting transfers of a value from a register into memory. It is stated that recomputation of a common subexpression is sometimes needed in order to minimize the number of registers to be used to evaluate an expression.

In the discussion, straight line programs are represented by dags. The problems looked at are:

1) given a dag D and a number of registers k, does a computation exist such that no values are recomputed and no more than k registers are used,
2) given a dag D and a number of registers k, does a computation exist where values may be recomputed and no more than k registers are used, and
3) given a dag D and a function L from the nodes of D into the set of names, does there exist a computation such that every node is computed, descendants are computed before their ancestors, and the value of a node is retained in the appropriate register as long as it is needed.

These problems are treated in mathmatical form and found to be polynomial complete.

Article 50

Wm. A. Wulf, R.K. Johnsson, C.B. Weinstock, S.O. Hobbs

The Design of an Optimizing Compiler

American Elsevier, New York, 1975

Compiler Implementation, Storage Minimization

This is an excellent description of the implementation of an optimizing compiler. It takes the design of the Bliss/11 compiler through all of its major phases:

1) lexical analysis, declaration processing, syntax analysis, and flow analysis.
2) determine shape, cost, and execution order of the code to be generated.
3) temporary storage allocation.
4) produce local optimizations.
5) "peephole" optimizations.

The descriptions of these phases are discussed in a manner which is easily understood by anyone who has had any experience with compiler implementation.

INDEX