PROGRAM OPTIMIZATION - THEORY AND PRACTICE

David B. Loveman
Ross A. Faneuf
Massachusetts Computer Associates, Inc.

Introduction

The conventional program optimization techniques employed by the ILLIAC FORTRAN compiler are general purpose, effective, and efficient. The underlying theory is applicable to FORTRAN and to other high level languages. A unique approach to the gathering of global set and use information about variables as well as careful software engineering of the algorithms has led to the construction of an effective source-to-source optimizer which performs constant propagation, constant computation, common subexpression elimination, reduction in strength, and invariant code motion.

We will first consider the type of information which must be gathered about a program, and how this information is used to perform optimization. Then we shall state the algorithm for globally computing the set and use information for program variables. Having discussed the science of optimization we shall turn to the engineering aspects and consider such topics as representation of programs, order of optimization transformations, and efficient computation of global use and set information.

Information Needed

In order to perform optimization we must have certain information available to us. This is of two types: information directly relating to the source text of the program and local pieces of the program, and information about the global structure of the program. The local information we need is:

- The program text in some representation convenient for manipulation, such as a list of statements with expressions in a tree form,
- Expressions placed in a convenient partly canonical form, (op, k, s_1, ..., s_n, e_1, ..., e_m) where "op" is the operation, "k" is a constant operand, "s_1" to "s_n" are the scalar operands in symbol table order, and "e_1" to "e_m" are the remaining operands which may be expressions, array references, etc., in any order. Note that local constant computation will be performed in producing this form. For example, 3+X*4+2+Y will be represented as (+,5,Y, (*,4,X)) .
- Conventional symbol table information such as whether a variable X is declared local or global,
- The language specified loop structure of the program (in FORTRAN we are interested in DO loops and not interested in loops constructed from IF and GOTO), and
- Flow of control information; a structuring of the program into flow blocks and their interconnections. Statements are tagged with flow block number and statement number (in lexical order).

This local information is quite conventional and easy to gather.

The global information required is more interesting:

- The "dominates" relation between flow blocks must be determined. For two flow blocks i and j , i dominates j if and only if every path from the program entry point to flow block j must pass through flow block i . This is a conventional definition of a conventional program graph relation which can be computed in a straightforward manner.
- The flow blocks are ordered in "flow-sequence" order as determined by the Earnest, Balke, and Anderson algorithm [9]. This algorithm assigns sequence numbers to flow blocks so that:
  1) every block will have a higher number than its predecessors, except for true backward transfers,
  2) every block will have a lower number than every other block which it dominates, and
  3) for any program looping structure, all and only those blocks in the loop will have numbers between those of the first and last blocks in the loop: thus to test whether a given flow block is within a given loop structure requires only two sequence number comparisons.
- Generation and use information about variables must be determined. For a variable X , we refer to "the p-graph of X" as the sum of information about uses and generations of X . In a later section, we will present the algorithm to compute p-graphs. For now we present

a general discussion of the information which is needed.

The fundamental category of information which results from p-graph analysis of a program is the determination of the "generation-class" membership of each appearance of each variable in the program. A "generation-class" of a variable in a program is the set of all those appearances of the variable which must represent the same value: this set will, of course, consist of one appearance at which the variable receives its value (this appearance is called "the generation"), and a number of appearances where the value is used, without being changed.

Consider a point in a program where two paths of control flow together, after which some variable is used one or more times, and assume that this variable receives values on each of the merging paths, before the merge. Each use after the merge will indeed belong to one or the other of the two generation-classes initiated by the generations before the merge, but in general it is impossible, at compile time, to tell which one; nonetheless, whichever one it is, all uses of the same variable after the merge, up until the next generation for that variable, must belong to the same generation-class (for any one passage of control at run time). For this reason, merge points are considered to be generations for any variable whose value is used after the merge before being reset, and a new generation-class is constructed for this pseudo-generation and the uses which depend from it. These merge generations are not, of course, represented by any code in the program.

We assume that p-graph information is represented in the program as follows:

- Every assignment to a variable is tagged with
  1) its generation class, which is a unique identification of this particular generation, and
  2) a list of all instances of the variable in the program which are uses of this generation.
- Every use of a variable is tagged with a pointer to its generation, and
- At appropriate merge points, pseudo-generations are produced which have the same characteristics as assignments, but which don't actually appear in the source program.

Optimization

With the available information, a number of program optimizations are performed. We present them here in a logical order, not necessarily the most efficient processing order:

- Linearize Array References. Multidimensional arrays are replaced by appropriately sized vectors and the addressing calculation is made explicit. For example, assuming row-wise storage, if A is declared to be a $5 \times 4 \times 3$ array, A will be replaced by a vector A^1 of 60 elements and each reference to A of the form A(I, J, K) will be replaced by A^1((I-1) * 12 + (J-1) * 3 + K) , or $A^1(12*I + 3 * J + K - 15)$. In making the addressing calculations explicit, they are exposed for later optimizations.
- Constant Propagation and Computation. For any assignment of a constant to a scalar variable X , all uses of this generation of X can be replaced by the constant. If the scalar X is a local variable and the assignment is not needed for a merge pseudo-generation, the assignment statement may be deleted. If all references to the scalar X are deleted, its definition may be removed from the symbol table. Some constant computation was performed when expressions were placed into canonical form. Constant computation is again performed during constant propagation whenever a variable use is replaced by a constant. Note that if this reduces the right hand side of an assignment to a constant, another constant propagation is necessary. It appears that constant propagation is a recursive process; in fact if the program is searched for assignments in block order we almost never produce a requirement for a new constant propagation where the assignment is "upstream" of where we now are. The occasional exception is stacked for processing when the primary scan is complete. It is important to note that, in a cross compiler, constant computation must be done using target machine arithmetic and not using host machine arithmetic.
- Dead Code Elimination. As a result of constant propagation some assignment statements and data declarations may be eliminated. It may be that after constant computation some DO loops can be determined as having exactly one iteration. In this case, the extraneous loop structure is eliminated (and also, perhaps, the loop variable). In some IF statements, the value of the predicate will now be TRUE or FALSE. This allows pruning of unreachable portions of the program. Since these transformations destroy the collected global information, they are all done together and the global information is recomputed.
- Scalar Propagation. Assignments of the form "X ← Y" for which only one generation of Y exists in the program can be eliminated and all uses of that generation of X can be replaced by Y .
- Invariant Code Motion. Computations which are invariant within a loop are moved to a point immediately before the loop entry. Loops are processed in lexical order and nests of loops are processed inside out: in other words in the lexical order of their end statements. A computation represented canonically as (op, X, Y) is loop invariant if the generators of X and Y are not within the loop. This test is simple because of the block order: find the block number i of the first block in the loop, the block number j of the last block in the loop, and the block number k of the block

containing the generation of A . The generation of X is not within the loop if k < i or k > j .

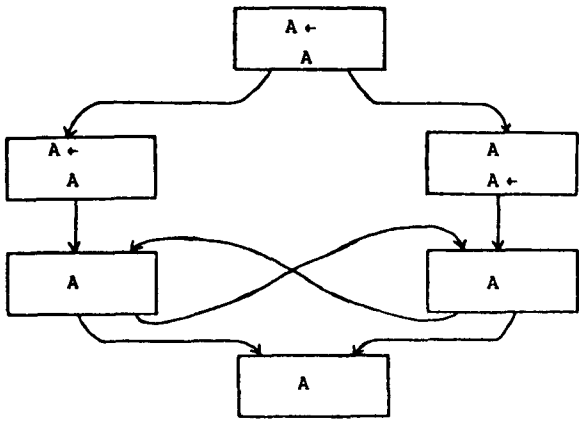
- Strength Reduction. Computations of the form I * L where I is the loop variable and L is loop invariant are candidates for strength reduction. The assignment $t \leftarrow I_{init} * L$, where I_{init} is the value I takes on the first iteration of the loop, is placed before the loop, the use of I * L is replaced by t , and the assignment t ← t + L is inserted at the end of the loop.
- Test Replacement. After strength reduction, if the loop variable has no explicit uses within the loop, replace the loop variable in the loop statement with the most frequently used variable generated by strength reduction. Incorporate its initial, increment, and final values into the loop statement.
- Folding. If for a generation "X ← expression" there is only one use of X , and if the variables of "expression" do not have generations before the use of X , the generation can be deleted and the use of X replaced by the expression. This is done only within a flow block since testing the conditions globally is difficult and the payoff, globally, is low.
- Common Subexpression Elimination. The program is scanned in block order, and statements are scanned bottom up looking for candidate computations. A candidate computation is an operation with at least one scalar operand X . Given a candidate, the set of uses of the generator of the variable X is searched to find a set of possible common subexpressions; these are computations containing a use of the generator of the variable X , and with the same operator as the candidate computation. The set of possible common subexpressions is pruned by comparing the second operands with the second operand of the candidate computation. We have now found a set of common subexpressions. All of these are eliminated in favor of the generated variable t and the computation "t ← cse", where cse is the candidate computation, is placed in an appropriate place. This assignment is placed in a block which is dominated by the generators of every variable in the cse, and which dominates every occurrence of the cse. There must be at least one such block, there may be several. The computation is put in the latest one (in block order) which is of lowest frequency (as measured by depth of loop nesting). The operations for which common subexpressions are computed are the common arithmetic, relational and logical ones, as well as functions (such as trigonometric) about which the system has knowledge. Since division is expensive, it is represented in the canonical form as a unary inversion operator. Thus common subexpression elimination will transform "...A/D ... B/D..." into "t ← 1/D ... A*t ... B*t ..." . The requirement for identical match is relaxed for the operations + and * . In these cases partial matches are allowed also. The search for a partial match is aided by the canonical form requirement that scalar appearances are ordered. Careful sign conventions in the canonical form allows cse to recognize, for example, that "... A-B ... B-A ..." can be written "t ← A-B ... t ... -t ..." . Because the check for identity occurs first, it is necessary to check generated statements for cse also. Consider the example:

				$t_3 \leftarrow A+B$
		$t_1 \leftarrow A+B+C$		$t_1 \leftarrow t_3+C$
		$t_2 \leftarrow A+B+D$		$t_2 \leftarrow t_3+D$
... A+B+C ...	⇒	... t_1 ...	⇒	... t_1 ...
... A+B+C ...		... t_1 ...		... t_1 ...
... A+B+D ...		... t_2 ...		... t_2 ...
... A+B+D ...		... t_2 ...		... t_2 ...

The second transformation would not occur unless the generated statements were also processed.

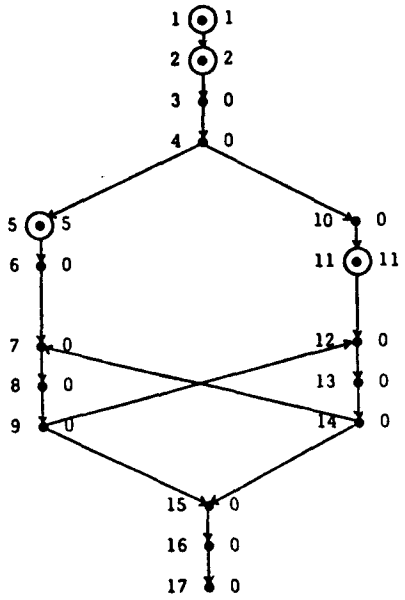
The P-Graph Algorithm

The p-graph algorithm is essential to the optimization process. This algorithm provides the generation class information for a given program variable. We shall first give an example of the algorithm applied to a variable in a simple program graph, then state the complete algorithm:

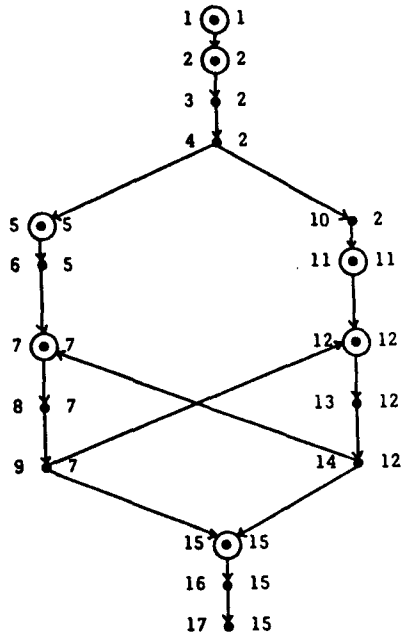


We logically expand this graph so that our attention is focused only on the variable A and flow structure specific to A . The program entry and exit nodes are made explicit (1 and 17), flow of control merge nodes are made explicit (7, 12, 15), block exit nodes (when there is more than one block successor) are made explicit (4, 9, 14), and each instance of A is made explicit (2, 3, 5, 6, 8, 10, 11, 13, 16). We wish to identify all generations and pseudo-generations and tag all uses with the appropriate generation. Program entry and all assignments to A act as generations; we indicate this by circling these nodes in the graph. Each node in the graph is numbered (on the

left). We wish to tag each node N (on the right) with a node number which is the number of the node corresponding to the generation of A which dominates node N . For each generation, the tag is clearly its own node number. For other nodes, initially, the tag is 0. This gives us the following graph:



The remaining problem is to determine the pseudo-generations which correspond to merges and to tag uses. This is done by pushing the tag (righthand side number) of each tagged node along the directed arcs to the node's successors and continuing along all paths until tagged nodes are reached. When a tag being pushed reaches a previously tagged node N , there are two possibilities: if N is tagged with its own node number, N is a generation and we are done; if N is tagged with some other number, N is a merge pseudo-generation and should be circled on the graph and tagged with its own node number to mark it as a generation. Upon completion of the algorithm every node is either circled and tagged with its own node number, or is tagged with a node number of a circled node, its unique most recent circled ancestor. These tags partition the graph nodes into equivalence classes where each circled node corresponds to an equivalence-class-generating event, either an assignment to A or a merge. It has been proved that the algorithm produces a unique minimal solution [1]. The resulting graph is:



It remains to present the p-graph algorithm itself. It should be noted that the p-graph algorithm does not depend on any particular characteristics of the program graph. The algorithm works perfectly well on an irreducible graph [10]; indeed, the example program graph is irreducible. Rather than code the algorithm in a particular programming language, we have expressed it using high level language constructs which should be familiar. A possible complication is the use of set operations on the sets S and NOTYET. This is done specifically to avoid issues about representation of sets; any representation will suffice.

We assume that we have initially:

- A constant N equal to the number of nodes,
- A vector TAG of N elements such that TAG(I) = I if the I^{th} node of the graph is circled; TAG(I) = 0 otherwise, for $1 \leq I \leq N$, and
- A vector S of N elements such that S(I) is the set of nodes which are immediate successors of node I , for $1 \leq I \leq N$.

The algorithm uses I as an integer variable to represent the node of current interest, J as an integer variable representing an immediate successor of I , Q as a boolean variable used as a flag to indicate whether another iteration through the graph is necessary, and NOTYET as a set of nodes still to be processed.

On completion of the algorithm N and S are unchanged and TAG(I) = I , if the I^{th} node is circled (generation or pseudo-generation); otherwise, TAG(I) = J , where the J^{th} node is the last circled ancestor on all paths to node I .

The p-graph algorithm is:

```
BEGIN: Q ← false;
       NOTYET ← { I | 1 ≤ I ≤ N and TAG(I) ≠ 0 };   // start with all circled nodes
       while NOTYET ≠ empty do
             choose I ε NOTYET;                     // select a node I to process
             NOTYET ← NOTYET - {I}                  // remove I from the set to be processed
             ( ∀J | J ε S(I) ) do                   // for all immediate successors J of I
                   if TAG(J) = 0 then               // if J is not tagged
                         TAG(J) ← TAG(I);           // tag it the same as I
                         NOTYET ← NOTYET ∪ {J};     // and add it to the set to be processed
                         goto NEXTJ;
                         endif;
                   if TAG(J) = TAG(I) then goto NEXTJ; endif;   //if J is tagged with
                   if TAG(J) = J then goto NEXTJ; endif;        // TAG(I) or J, ignore it
                   TAG(J) ← J;                      // otherwise J is a merge, circle J
                   Q ← true;                        // set flag to indicate another pass
                                                    // needed
             NEXTJ: end all;
             end while;
       if Q then                                    // if another pass is needed
             for   I ← 1 to N do                    // erase all tags except those
                   if TAG(I) ≠ I then TAG(I) ← 0; endif;   // corresponding to circling
                   end;
             goto BEGIN;
             endif;
```

Optimization Engineering

We have seen how to use p-graph techniques to perform conventional program optimization. We must now consider, at least briefly, the issue of whether the transformations and p-graph techniques can be performed efficiently enough to be of practical utility. A first necessary requirement is an appropriate choice for internal representation of programs. We shall say no more than that it has proved valuable to have a rich internal structure which allows, for example, forward or backward scans through the program test; iteration over all instances of the variable X , rapid location of loop structures, etc.

An important issue is the order in which program transformations are executed. The following order has proved valuable:

1) Constant propagation with constant computation and dead code elimination done as need for them is detected. Constant propagation may result in a DO loop being executed only once. If this is the case, elimination of the loop structure may cause a constant value to be propagated for the loop index. Since the program structure is changing, p-graphs are generated only for those variables which are candidates for constant propagation. Following completion of constant propagation the flow structure of the program will remain constant. The p-graphs which have been generated and which are still valid are kept, and p-graphs for the other scalar variables in the program are generated.
2) Array references are expanded and linearized. Since most likely candidates for strength reduction come from array addressing expressions, there is a slight savings in incorporating strength reduction and test replacement here.
3) Common subexpression elimination is done next, along with scalar propagation, which helps identify more common subexpressions.
4) Invariant code motion is done after common subexpression elimination since common subexpressions which are loop invariant will be moved out of loops by the elimination process. Also eliminating common subexpression decreases the number of potential candidates for code motion.
5) Finally there is a clean up phase which includes folding and elimination of dead variables not previously eliminated during constant propagation.

The most important area for good software engineering is that of p-graph generation. Many special cases are detected for which p-graph generation is trivial. This leaves only a few cases for which the complete algorithm is needed. For example all generated variables are of one of two types: single generation variables for which all uses have the same generation, or iterated variables, which have two generations and one merge, and all uses are generated by the merge. Variables introduced by strength reduction are of this latter type.

All instances of a variable are separated into uses and generations. The rich structure of the intermediate language makes this an easy operation. An argument to a subprogram is a use and a generation, and every subprogram call is a generator of every common variable, in the absence of any evidence to the contrary. At any time in the processing, if there are no uses, or if all uses have been tagged, the p-graph is done. Flow blocks containing instances are marked, those containing one generation are tagged with that generation at block exit; those containing more than one generation are processed locally in statement serial order and the tag of the last generation is the block exit tag. If all uses within a block are tagged by generations within the block, the block is no longer a use block. An explicit entry generation may not be needed if all uses are in blocks dominated by generations. The dominates relation is represented by a bit matrix. A bit vector corresponding to those blocks containing generations

is easy to prepare. For each use, the appropriate row from the matrix is "anded" with the generation vector. If the result vector is non-zero, the use is dominated by some generation. For common variables there may be no generation. In this case the entry generation is the generation for all uses. At this point if there is only one generation we are done, otherwise we perform the p-graph algorithm.

There are various improvements in the p-graph algorithm that have been made to make it both more efficient and useful for the propagation of other property sets, but we shall not discuss these improvements here. The algorithm may generate spurious merges, pseudo-generations which in fact dominate no uses. These are eliminated from the p-graph.

An interesting case of globally initialized local variables occurs in some versions of FORTRAN. A typical programming practice is to call a routine with a special parameter the first time to indicate that initialization of local static storage is required. For example:

```
      SUBROUTINE WALDO ( I, A, B )
      IF ( I .NE. 0 ) GO TO 10
      J = 2
      X = 3.5
10    —
```

In this example J and X are local static variables. WALDO is called the first time with I = 0 to initialize J and X and with I ≠ 0 every other time. This case can be detected since J and X have only one generation, and the entry generation, and all uses of J and X are of the merge of those two generations. The constant values can then be propagated.

Finally we must make some comments on efficiency. First, using p-graph techniques, any program which can be parsed can be optimized. Unlike other techniques such as interval analysis, there is no requirement that the program graph be of any special form, such as irreducible. Second, the cost does not appear to be prohibitive. There is a substantial cost involved in invoking optimization at all, and this cost seems to increase fairly rapidly until a program size of about 10 blocks is reached. Beyond 10 blocks however the cost increases very slowly so that for (ILLIAC FORTRAN) programs of 500 to 1000 statements or more the cost of optimization is in fact less than the cost of parsing the program.

Acknowledgements

The initial description of p-graphs appeared in "Representation of Algorithms" by Shapiro and Saint [1]. This document also included the initial description of the p-graph algorithm by Stephen Warshall and a proof by Robert Millstein that the algorithm produces a unique minimal solution. An optimizer functioning as described in this paper was designed and implemented by Ross Faneuf for use in the ILLIAC FORTRAN compiler [2] and is currently operational. During this implementation suggestions were made to improve the efficiency of the p-graph algorithm by Leslie Lamport [3]. Recent work by Ben Wegbreit at Harvard [4] and Michael Karr of Mass. Computer Associates [5], [6], [7] has extended and generalized the basic p-graph techniques. An optimizer similar to the one described here but incorporating significant new results is being implemented in a "language laboratory" to allow experimental optimization of a FORTRAN like language [8].

Bibliography

[1] Shapiro, Robert M. and Saint, Harry. "The Representation of Algorithms", Applied Data Research, Inc., Final Technical Report. RADC-TR-69-313, Volume II, Rome Air Development Center, Sept., 1969.

[2] Massachusetts Computer Associates, Inc. "Sixth Semi-Annual Technical Report (14 July 1972 - 13 February 1973) for the Project Compiler Design for the ILLIAC IV", CADD-7302-2011, February, 1973.

[3] Lamport, Leslie. "A Refinement of the Warshall P-Graph Completion Algorithm", ILLIAC FORTRAN Compiler Project Internal Memo, Mass. Computer Associates, Inc., November 9, 1972.

[4] Wegbreit, Ben. "Property Extraction in Well-Founded Property Sets", Center for Research in Computing Technology, Harvard University, February, 1973.

[5] Karr, Michael. "On Affine Relationships Among Variables of a Program", Mass. Computer Associates, Inc., CA-7402-2811, February, 1974.

[6] Karr, Michael. "Proving Inequalities", Mass. Computer Associates, Inc., CA-7406-1011, June, 1974.

[7] Karr, Michael. "The P-Graph Algorithm", in preparation.

[8] Loveman, David, Sattley, Kirk and Bearisto, David. "Development of Compiler Optimization Techniques", Mass. Computer Associates, Inc., CADD-7407-2311, July, 1974.

[9] Earnest, C.P., Balke, K.G., and Anderson, J. "Analysis of Graphs by Ordering of Nodes", Journal of the ACM, Vol. 19, No. 1, January, 1972.

[10] Schaefer, Marvin. A Mathematical Theory of Global Program Optimization, Prentice-Hall, 1973.