The Program Dependence Graph and Its Use in Optimization

JEANNE FERRANTE
IBM T. J. Watson Research Center
KARL J. OTTENSTEIN
Michigan Technological University
and
JOE D. WARREN
Rice University

In this paper we present an intermediate program representation, called the *program dependence graph* (*PDG*), that makes explicit both the data and control dependences for each operation in a program. Data dependences have been used to represent only the relevant data flow relationships of a program. Control dependences are introduced to analogously represent only the essential control flow relationships of a program. Control dependences are derived from the usual control flow graph. Many traditional optimizations operate more efficiently on the PDG. Since dependences in the PDG connect computationally related parts of the program, a single walk of these dependences is sufficient to perform many optimizations. The PDG allows transformations such as vectorization, that previously required special treatment of control dependence, to be performed in a manner that is uniform for both control and data dependences. Program transformations that require interaction of the two dependence types can also be easily handled with our representation. As an example, an incremental approach to modifying data dependences resulting from branch deletion or loop unrolling is introduced. The PDG supports incremental optimization, permitting transformations to be triggered by one another and applied only to affected dependences.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers, optimization*

General Terms: Algorithms, Languages, Performance

Additional Key Words and Phrases: Data flow, dependence analysis, intermediate program representation, internal program form, vectorization, parallelism, node splitting, code motion, loop fusion, slicing, debugging, incremental data flow analysis, branch deletion, loop unrolling

1. INTRODUCTION

This paper introduces a program representation, called the *Program Dependence Graph* or *PDG*, that provides a unifying framework in which previous work in program optimization may be applied. We present a new incremental data flow algorithm that operates directly on the PDG. This algorithm is important not

Karl Ottenstein's work was supported in part by the National Science Foundation under grants DCR-8203487, DCR-8404463, DCR-8404909, and DCR-8511439.

Authors' addresses: Jeanne Ferrante, IBM T. J. Watson Research Center, Hawthorne, H2-B54, P.O. Box 704, Yorktown Heights, NY 10598; Karl J. Ottenstein, Department of Computer Science, Michigan Technological University, Houghton, MI 49931; Joe D. Warren, Department of Computer Science, Rice University, Houston, TX 77251.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1987 ACM 0164-0925/87/0700-0319 $01.50

only because of its efficiency, but also because unlike other incremental flow algorithms, it permits *incremental optimization* as the data flow information is updated. Work by others in the area of code motion, vectorization, program understanding, and software engineering can be unified by being expressed in terms of the PDG.

The PDG makes explicit both the data and control dependences for each operation in a program. Data dependence graphs have provided some optimizing compilers with an explicit representation of the definition-use relationships implicitly present in a source program [31, 36]. A control flow graph [1, 3] has been the usual representation for the control flow relationships of a program; the control conditions on which an operation depends can be derived from such a graph. An undesirable property of a control flow graph, however, is a fixed sequencing of operations that need not hold. The program dependence graph explicitly represents *both* the essential data relationships, as present in the data dependence graph, and the essential control relationships, without the unnecessary sequencing present in the control flow graph.[1] These dependence relationships determine the *necessary* sequencing between operations, exposing potential parallelism.

Since dependences in the PDG connect computationally relevant parts of the program, many code improving transformations require less time to perform than with other program representations. A single walk of these dependences is sufficient to perform many optimizations. Since both kinds of dependences are present in a single form, transformations like vectorization can treat control and data dependence *uniformly*. Program transformations such as code motion, which require *interaction* of the two types of dependences, can also be easily handled by our single graph. In addition, the hierarchical nature of the PDG allows large sections of the program to be summarized. It is thus the basis for efficient algorithms for many reordering transformations [50].

Our motivation in developing the PDG has been to develop a program representation useful in an optimizing compiler for a vector or parallel machine. Such a compiler must perform both conventional optimizations as well as new transformations for the detection of parallelism. The PDG is of interest for conventional machines as well, because of the efficiency with which many powerful optimizations may be performed. (The reader is forewarned, though, that the methods discussed in this paper cannot be directly incorporated into an existing compiler; after interprocedural data flow analysis, a PDG must be built from whatever initial intermediate is produced by the existing front-end.)

The remainder of the paper will address each of these issues in turn: related work in dependence-based program representations; construction of the PDG; a sampling of applications of the PDG; and a detailed description of a new incremental data flow algorithm for use when performing branch deletion, loop unpeeling, and loop unrolling. An appendix contains the basic graph theoretic terminology used.

[1] Of course, determining the exact control conditions under which an operation is executed (like determining the exact definition-use relationships) is an unsolvable problem. However, we can conservatively determine all possible control conditions, just as all possible definition-use relationships can be determined by data flow analysis.

2. RELATED WORK

Much related work has been performed over the past ten years in the area of dependence-based program representations. Dennis' work [18] opened up the area of data flow computation [19]. Most representations in that area treat all dependences as data dependences, control dependences being converted as necessary [17]. (Some representations, however, do treat control dependence differently [49].) Unnecessary statement orderings are eliminated in data flow machine graphs, exposing low-level parallelism. Yet, due to the distribution of control operators throughout the data dependence edges, both data and control become too fragmented for the convenient application of conventional optimizations.

The program plans [52] of the Programmer's Apprentice project [51] represent control and data dependences in a modularized form in which loops have been converted to recursion. The goals of that project involve program understanding to aid modification; the form used does not appear amenable to traditional optimizations.

The *data dependence graphs* [48] used in the Illinois vectorizer "Parafrase" [41] are designed for the hierarchical analysis of dependence relations in programs; those graphs are threaded through a syntactic representation of the program as a multilist of tokens. This threaded approach complicates optimization; it is motivated by the Parafrase design goal to have each optimization be an independent source-to-source pass. Further work developing the ideas of dependence depth and loop-carried dependence [7, 8] for vectorizing transformations [9] has been carried out at Rice University. Many of the results of these two groups are incorporated into the PDG. The control dependence representation of the PDG is one major advantage over the data dependence graphs, in that the need to convert control dependence into guards [8, 41] is eliminated and vectorization is in some cases reduced to simple path questions [50].

IF1 [46] is a proposed intermediate for applicative languages such as SISAL [34]. IF1 is a hierarchical graph form in which data dependences are explicit at the lowest levels and are implicit between so-called "compound nodes," representing control structures, and their subgraphs. It is simpler than the PDG because it need not deal with side effects and arbitrary control flow; it is not readily apparent how IF1 might be extended for use with imperative languages. IF1 has been used as the basis for partitioning and scheduling functional programs for multiprocessors [44, 45].

The *Data Flow Graph* [36, 37] represents global data dependence at the operator level (called the *atomic* level in [31]). Transformations that involve both control and data dependence cannot be specified in a consistent manner with this form, however, since control is represented by a conventional control flow graph. The *Extended Data Flow Graph* [23] represents control dependence consistently with data dependence, but can only represent "structured" programs.[2] The Program Dependence Graph, described here, eliminates this restriction on control flow.

[2] We call a program *structured* if it is built of blocks, loops, and conditionals, each of which is single entry and single exit.

3. THE PROGRAM DEPENDENCE GRAPH

The PDG represents a program as a graph in which the nodes are statements and predicate expressions (or operators and operands) and the edges incident to a node represent both the data values on which the node's operations depend and the control conditions on which the execution of the operations depends. Nodes representing statements and predicates are sufficient for some transformations such as vectorization and simplify our illustrations in this paper. For almost all other optimizing transformations, nodes represent operators and operands. The set of all dependences for a program may be viewed as inducing a partial ordering on the statements and predicates in the program that must be followed to preserve the semantics of the original program.

Dependences arise as the result of two separate effects. First, a dependence exists between two statements whenever a variable appearing in one statement may have an incorrect value if the two statements are reversed. For example, given

$$\begin{array}{ll} A = B * C & S1 \\ D = A * E + 1 & S2 \end{array}$$

$S2$ depends on $S1$, since executing $S2$ before $S1$ would result in $S2$ using an incorrect value for A. Dependences of this type are *data* dependences. Second, a dependence exists between a statement and the predicate whose value immediately controls the execution of the statement. In the sequence

```
if (A) then        S1
    B = C * D      S2
endif
```

$S2$ depends on predicate A since the value of A determines whether $S2$ is executed. Dependences of this type are *control* dependences.

In presenting the construction of the PDG below, we emphasize the construction of the control dependence subgraph. Control flow analysis is used in building the control dependence subgraph. We show methods to compute the exact control dependences and then present an approximation that preserves more of the original control flow structure. The construction of the data dependence subgraph is based on previous work [36, 37, 41]. Data flow analysis is used to compute the sets of definitions that reach each basic block for use in building the data dependence subgraph. We also show how aliasing and side effects are accommodated in the PDG. The remaining subsections discuss the variety of program views supported by the PDG and some practical considerations.

3.1 Control Dependence

In this section, we define control dependence in terms of a control flow graph and dominators [1, 3]. The Appendix contains the basic graph theoretic terminology used.

Definition 1. A *control flow graph* is a directed graph G augmented with a unique *entry* node START and a unique *exit* node STOP such that each node in the graph has at most two successors. We assume that nodes with two successors have attributes "T" (*true*) and "F" (*false*) associated with the outgoing edges in

the usual way. We further assume that for any node N in G there exists a path from START to N and a path from N to STOP.

Definition 2. A node V is *post-dominated* by a node W in G if every directed path from V to STOP (not including V) contains W.

Note that this definition of post-dominance does not include the initial node on the path. In particular, a node never post-dominates itself.

Definition 3. Let G be a control flow graph. Let X and Y be nodes in G. Y is *control dependent* on X iff

(1) there exists a directed path P from X to Y with any Z in P (excluding X and Y) post-dominated by Y and
(2) X is not post-dominated by Y.

If Y is control dependent on X then X must have two exits. Following one of the exits from X always results in Y being executed, while taking the other exit may result in Y not being executed. Condition 1 can be satisfied by a path consisting of a single edge. Condition 2 is always satisfied when X and Y are the same node. This allows loops to be correctly accommodated by our definition. The transitive closure of our definition corresponds to the notion of the range of a branch given in [55].

When applied to a loop in the control flow graph, our definition of control dependence determines a strongly connected region (SCR) of control dependences whose nodes consists of predicates that determine an exit from the loop. The other nodes in the control flow graph loop not in the SCR of control dependences lie on some path of control dependence edges from a node in the SCR. Intuitively, these correspond to the body of the loop. For example, see the DO loop and its corresponding PDG in Figure 8. In a control flow graph, nested loops appear as nested SCRs and the hierarchy must be discovered. With the PDG, nested loops appear as distinct SCRs with a control dependence edge between the outer loop and each immediate inner loop. Loops at the same level appear as SCRs with a common ancestor region.

We show in Figure 1 a control flow graph and its corresponding control dependence subgraph.[3] *Region nodes R1* through *R6* and *Entry* have been inserted to summarize the set of control dependences for each node, as explained in the following section. Control dependence edges are represented here and throughout this paper as dashed lines. (Note that nodes 1 and 7 are control dependent on program entry; they could thus be executed in parallel, if no data dependences exist between them. This relationship is not immediately apparent from the control flow graph.)

Control dependences to nodes at the operator level are determined by the corresponding control dependences at the statement and predicate level, which are simply extended to all contained operators.

3.1.1 *Determining Control Dependence.* The first step in determining control dependences is the construction of the post-dominator tree for an augmented

[3] The special nodes START and STOP do not appear in the control dependence subgraph, as these empty blocks were added to the control flow graph for analysis purposes only.

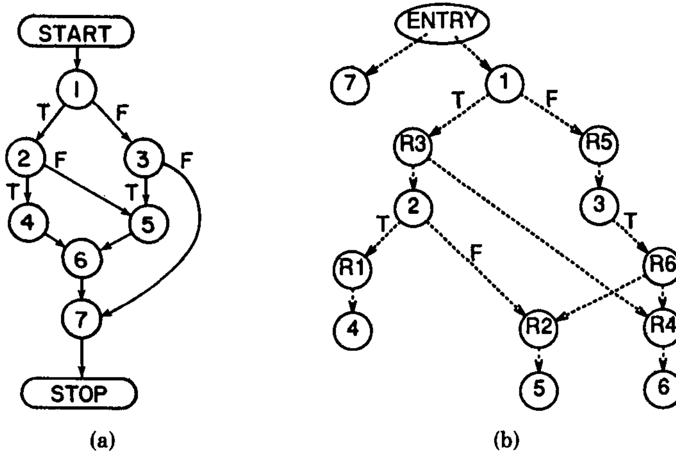

Fig. 1. A control flow graph and its control dependence subgraph.

control flow graph. We augment the control flow graph with a special predicate node ENTRY that has one edge labeled "T" going to START and another edge labeled "F" going to STOP. ENTRY corresponds to whatever external condition causes the program to begin execution. Computing post-dominators in the control flow graph is equivalent to computing dominators [1] in the reverse control flow graph. Dominators in the reverse graph can be computed quickly by using the method of [33] to construct the dominator tree. Using this method, the post-dominator tree of a graph can be constructed in time $O(N\alpha(N))$, where N is the number of nodes (basic blocks) in the control flow graph.[4] Figure 2 shows the post-dominator tree for the augmented control flow graph based on Figure 1.

Given the post-dominator tree, we can determine control dependences by examining certain control flow graph edges and annotating nodes on corresponding tree paths. Let S consist of all edges (A, B) in the control flow graph such that B is not an ancestor of A in the post-dominator tree (i.e., B does not post-dominate A). Each of these edges has an associated label "T" or "F". In our example, $S = \{(\text{ENTRY}, \text{START}), (1, 2), (1, 3), (2, 4), (2, 5), (3, 5)\}$. The control dependence determination algorithm proceeds by examining each edge (A, B) in S. Let L denote the least common ancestor of A and B in the post-dominator tree. By construction, we cannot have L equal B.

CLAIM. *Either L is A or L is the parent of A in the post-dominator tree.*

PROOF. Let X denote A's parent in the post-dominator tree. By construction, X is not B. We first show that X post-dominates B. Suppose it doesn't. Then there would be a path from B to STOP that does not pass through X. But then adding edge (A, B) to this path yields a path from A to STOP that does not pass through X (since, by construction, X is not B). This contradicts the fact that X post-dominates A. Since X post-dominates B, it must be an ancestor of B in the

[4] This can also be stated as $O(E\alpha(E))$ since E, the number of edges, is at most twice N for a program graph.

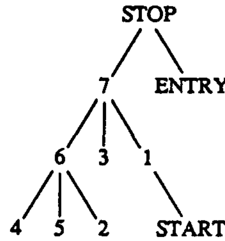

Fig. 2. Post-dominator tree for augmented control flow graph.

Edge in S examined	Nodes marked	Control dependent on	Label
(ENTRY, START)	START, 1, 7	ENTRY	T
(1, 2)	2, 6	1	T
(1, 3)	3	1	F
(2, 4)	4	2	T
(2, 5)	5	2	F
(3, 5)	5, 6	3	T

Fig. 3. Control dependences determined for each edge in S.

post-dominator tree. If X, A's immediate post-dominator, post-dominates B, then the least common ancestor of A and B in the post-dominator tree must be either X or A itself. □

We now consider these two cases for L, and show that one method of marking the post-dominator tree with the appropriate control dependences accommodates both cases.

Case 1. L = parent of A. All nodes in the post-dominator tree on the path from L to B, including B but not L, should be made control dependent on A.

Case 2. $L = A$. All nodes in the post-dominator tree on the path from A to B, including A and B, should be made control dependent on A. (This case captures loop dependence.)

It should be clear that, given (A, B), the desired effect will be achieved by traversing backwards from B in the post-dominator tree until we reach A's parent (if it exists), marking all nodes visited before A's parent as control dependent on A. This single traversal handles both cases above since, for Case 1, A will not be on the path to L and will thus not be marked. After all edges in S have been examined, all control dependences have been determined.

The table in Figure 3 shows the control dependences that would be determined by examining each of the edges in the set S for the graph in Figure 1. As there are no loops in that graph, all dependences are determined according to Case 1 above.

The correctness of the construction follows directly from the definition of control dependence. Referring back to this definition, for any node M on the

path in the post-dominator tree from (but not including) L to B, (1) there is a path from A to M in the control flow graph that consists of nodes post-dominated by M, and (2) A is not post-dominated by M. Condition (1) is true because the edge (A, B) gives us a path to B, and B is post-dominated by M. Condition (2) is true because A is either L, in which case *it* post-dominates M, or A is a child of L not on the path from L to B.

We now analyze the time requirements of this algorithm. The post-dominator tree can be constructed in time $O(N\alpha(N))$, where N is the number of nodes in the control flow graph. An edge (A, B) is determined to be in S in constant time, if post-dominator information is retained in bit vectors. Thus, S is determined in time $O(E)$ or, equivalently, $O(N)$. For each edge (A, B) in S examined in the algorithm, marking the appropriate nodes in the path from L to B with a single control dependence can be done in time proportional to the worst-case path length, namely the height of the post-dominator tree,[5] $O(N)$. Total marking time is thus $O(N^2)$. Finally, walking the post-dominator tree and building a control dependence subgraph can be done in N steps, where each step adds at most N edges. Thus, this phase of construction requires time at most $O(N^2)$.

The next step of the PDG control dependence subgraph construction is the addition of *region nodes* to summarize the *set* of control conditions for a node and group all nodes with the same set of control conditions together. This is accomplished so that if one set of control dependences contains another, the contained set will be factored into the control dependence representation for the containing set. Region nodes are also inserted so that predicate nodes will have only two successors, as in the control flow graph, hierarchically organizing the control dependences. Region node insertion may be viewed as a limited form of common subexpression elimination for control dependences and an extended form of basic block construction. Region nodes are created for common control dependence subsets that are factored out of the set of control dependences for a particular node. In the discussion below, two nodes are said to have the same control dependence predecessors if each has control dependence edges from exactly the same nodes and the corresponding edges have the same labels on them: "*T*", "*F*", or none.

First, we consider the set CD of control dependence predecessors of *each* nonregion node that has other than a single unlabeled control dependence predecessor. A region node R is created for CD, and each node in the graph whose set of control dependence predecessors is CD is made to have only the single control dependence predecessor R. Finally, R is given CD as its set of control dependence predecessors. To carry out this factoring, if R's set of control dependence predecessors is a subset of the set of control dependence predecessors for some other node, that node is made directly control dependent on R in place of the nodes in CD. (This approach does not *completely* factor the dependences, since there could well be two nodes for which the intersection of control dependences is not equal to the dependences of either node. But we are only interested

[5] The worst-case tree would arise from a control flow graph of N nodes consisting of a sequence of $N/2$ if-then's. The post-dominator tree would be the degenerate tree with $N/2$ nodes, hence the $O(N)$ height. We would not expect such trees in practice.

in determining regions that are subregions of one another to capture nesting properties.)

To implement this first step of region node insertion, we use a hash table that maps sets of control dependences to generated region nodes. We hash the set of control dependences for a node to determine if a region node for that set already exists. By keeping the control dependence lists canonically ordered, set comparison can be done in $O(N)$ time. Thus, the cost of checking the hash table for an equivalent region node is $O(N)$. In practice, we expect the number of control dependences to be bounded by a constant, yielding $O(1)$ time for finding an equivalent region node.

An efficient factoring of dependences is based on the observation that, by construction, any nodes having a proper containment of their sets of control dependences must be adjacent to one another on some path in the post-dominator tree. Thus, we implement region node insertion by means of a postorder traversal of the post-dominator tree to assure that all children in the post-dominator tree are visited before their parent. As each node N is visited, we check the hash table for an existing region node with the same set CD of control dependences.[6] If none exists, we create a new region node R whose predecessors are CD and enter R into the hash table. R is made to be the only control dependence predecessor of N. Next, we compute the intersection INT of CD with the set of control dependences for each immediate child of N in the post-dominator tree in $O(N)$ time [$O(1)$ expected in practice]. If the intersection INT equals CD, then the corresponding dependences are deleted from the child and replaced with a single dependence on R. If every control dependence of the *child* is in the intersection INT, then the corresponding dependences are deleted from R and replaced with a single dependence on the child's control predecessor. (The increase in graph size caused by region node insertion does not increase the time required for this process, since insertion does not increase the size of our work list, the remaining nodes in the post-order traversal of the post-dominator tree.) Since every node in the post-dominator tree is examined twice (once as a child and once as a parent), this step requires $O(N^2)$ time in the worst case with $O(N)$ time expected in practice.

Figure 4(a) shows the partial PDG control dependence subgraph after having visited nodes 4, 5, and 2 (introducing R1 through R3) and *just after* R4 has been created for node 6. The set CD for 6 is $\{3T, 1T\}$. There is a nonempty intersection with the control dependence set for child 5, namely $\{3T\}$. The intersection of CD with the control dependence set for child 2 is $\{1T\}$, corresponding to R3. Thus, the final step of visiting node 6 will be to remove the edge (1, R4) labeled "T", replacing it with the edge (R3, R4) having no label. Figure 4(b) shows the graph that results from visiting all nodes.

The second part of region node insertion requires a pass over the graph to make sure that each predicate node has a unique successor for each truth value. For any predicate node P in the control dependence subgraph which has multiple control dependence successors with the same associated label, L, we create a

[6] In fact, the hash table entry contains a list of the original control dependences even though the control dependence predecessors of the node pointed to may now include other region nodes due to factoring.

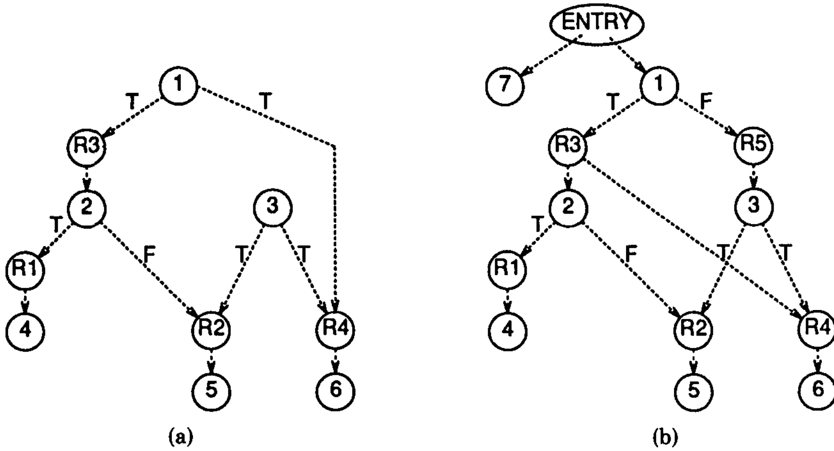

Fig. 4. Region node insertion.

region node R. Each node in the graph that had control dependence predecessor P with the label L is made to have the single control dependence predecessor R. Finally, R is made to be the single control dependence successor of P with the same label. Since, in the worst case, each predicate may have $O(N)$ successors, the total time for this step is $O(N^2)$. As with the previous pass, we expect $O(N)$ time in practice.

In Figure 4, node 3 has "T" edges to both R2 and R4. A new region node R6 has been introduced in Figure 1(b) to summarize these dependencies.

Region node insertion, as described, can be done in worst case time $O(N^2)$. The entire control dependence subgraph construction can thus be accomplished in time $O(N^2)$.

3.1.2 *An Approximation to Control Dependence.* Regenerating the original control flow graph from the control dependence subgraph may be of interest with a source-to-source translator; it also has implications for straightforwardly generating sequential code. Unfortunately, the reconstruction of control flow from the exact control dependence subgraph may be difficult. Consider the control flow graph in Figure 1 and its corresponding control dependence subgraph. Node 6 is control dependent only on node 1 and node 3. It is not obvious how to generate sequential code that places node 6 appropriately in the control flow graph without duplicating nodes or adding Boolean tests. In fact, this case can be handled by the methods of [22], which can generate control flow from reducible [1] or well-structured control dependence subgraphs without duplicating code or adding extra Boolean tests. In this section, we present an approximation to the exact control dependence subgraph from which it is easy to generate efficient control flow in all cases. This new graph, in addition, provides a much better approximation to control dependences than that provided by the control flow graph since, in the case of structured programs, it is identical to the exact control dependence subgraph.

The approximation we present will be based on the notion of single entry, single exit regions in the control flow graph called hammocks [27].

Definition. Let G be a control flow graph for program P. A *hammock* H is an induced subgraph of G with a distinguished node V in H called the *entry node* and a distinguished node W not in H called the *exit node* such that

(1) All edges from $(G - H)$ to H go to V.
(2) All edges from H to $(G - H)$ go to W.

The following theorem highlights one of the crucial properties of hammocks with respect to control dependence.

THEOREM. *Let G be a control flow graph for program P. Let H be a hammock of G. If X is in H and Y is in $(G - H)$, then Y is not control dependent on X.*

PROOF. Assume Y is control dependent on X. If Y is the exit node of H, then Y post-dominates X (all paths out of H leave through the exit node). This contradicts the assumption of control dependence. If Y is not the exit node of H, let P be any path from X to Y given by the definition of control dependence. Let W be the exit node of H. W is in P since W post-dominates X. By the definition of control dependence, Y also post-dominates W. This implies Y post-dominates X, which contradicts the assumption of control dependence. □

To produce a control dependence representation that allows easy generation of sequential code, a modified notion of *region node* and control dependence edge is required. In Section 3.1.1, a region node gathered together all statements (operators) that execute under the same conditions. If any one statement in a region is executed, so are all other statements. This is similar to the notion of a basic block. A region node there, though, did not incorporate any information about the control successors of the region. Just as a basic block has a set of statements and some number of exit edges, a region node here consists of two parts, a set of hammocks and some number of exit edges. The set of hammocks is a set of single entry, single exit control structures that can be generated in any order consistent with the data dependences between hammocks, as can the set of statements in a basic block. The exit edge(s) correspond to an absolute or conditional GOTO generated at the end of the set of hammocks, exactly as in a basic block. In what follows, we distinguish between control dependence edges whose successors are subhammocks and control dependence edges that are exit edges. Exit edges will be indicated by double lines.

In [27], an algorithm is presented for recognizing the hammocks[7] of a graph in $O(NE)$ operations, where N is the number of nodes and E the number of edges in the graph. Since E is at most $2N$ for a control flow graph, this bound is $O(N^2)$. The construction of the approximate control dependence subgraph consists of four steps. First, we detect hammocks in the control flow graph and mark the entry and exit nodes of each hammock. Second, we insert a region node as the

[7] The definition of hammock given here differs slightly from that of [27]. In particular, our definition of hammock includes single exit, single entry loops. In order to have loops included in the definition of [27], we transform all loops by inserting a dummy node as the single predecessor of the source of the back edge of the loop. All of the original predecessors of the source of the back edge are made predecessors of this dummy node (later referred to as a region node). In addition, we also insert a dummy node as a successor of each exit from the loop, with all of the original successors made successors of the dummy node. This transformation to the control flow graph allows us to use the definition of [27] directly.

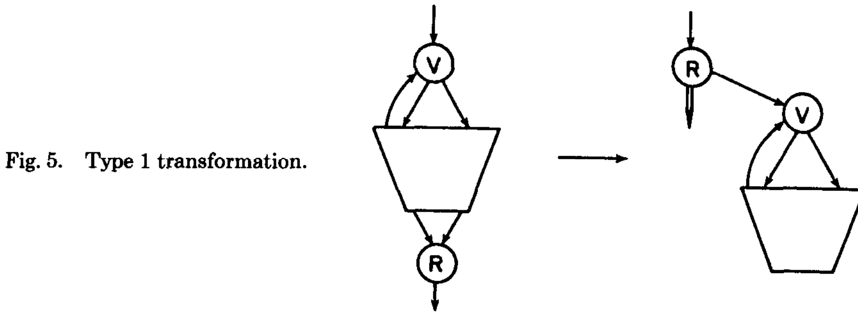

Fig. 5. Type 1 transformation.

single successor of STOP. Next, we convert each predicate node in the graph to a region node containing the empty set of subhammocks and whose exit edges are identical to those of the predicate. Finally, we apply one of the following transformations to each of the hammocks to construct the approximate control dependence subgraph. These transformations are applied in inverse topological order of exit nodes, with hammocks having a common exit node processed in inverse topological order of entry nodes.

Two types of transformations are needed to construct the approximate control dependence subgraph from the control flow graph. The first type of transformation is applied when the exit node of a hammock has all of its predecessors in the hammock. By processing the nodes in inverse topological order, and inserting a region node as the successor of STOP, we guarantee that the exit node of a hammock of this type will always be a region node. The tranformation consists of changing all of the predecessors of the hammock's entry node V which originate *outside* the hammock to predecessors of the exit node, R, and making the entry node a subhammock of the exit node. All edges previously incident on the exit node R are then deleted. This transforms all remaining hammocks with exit V to hammocks with exit R. We refer to hammocks of this type as type one hammocks and transformations of this type as type one transformations (see Figure 5). Edges to subhammocks are indicated by single arrows, and exit edges by double arrows.

The second type of transformation is applied to hammocks whose exit nodes have some predecessors outside the hammock. In this transformation, a new region node R is created and the predecessors of the entry node outside the hammock are made predecessors of R instead. Next, the entry node V is made a subhammock of R and an exit edge is inserted from R to W. Finally, each edge originating in the hammock and incident to the exit node is deleted. We refer to hammocks of this type as type two hammocks and transformations of this type as type two transformations (see Figure 6). In Figure 7, we show the approximate control dependence subgraph for the control flow graph in Figure 1.

Our reasons for suggesting these hammock-based transformations are as follows. For structured control flow graphs, the resulting control dependence graph always reflects the exact control dependences. More importantly, the resulting control dependence graph allows the quick and easy generation of sequential code. To generate code corresponding to a hammock, we simply generate code corresponding to its region nodes in a topological order imposed by control

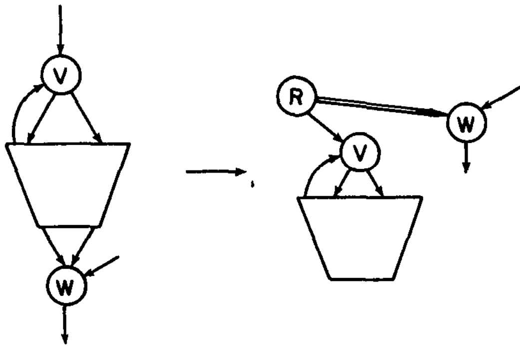

Fig. 6. Type 2 transformation.

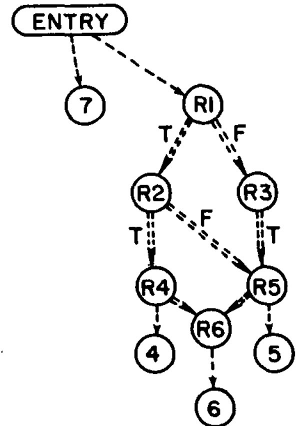

Fig. 7. Control dependence subgraph constructed using hammocks.

dependences between these nodes. To generate code corresponding to a region node, we generate code corresponding to the set of hammocks associated with the region node in a topological order imposed by the data dependences between hammocks. Using this control dependence graph obtained by transformations to hammocks, we thus trade a more accurate set of control dependences (for unstructured graphs) for the ability to easily generate sequential code.

3.2 Data Dependence

3.2.1 *Determining Data Dependence.* The construction of the data dependence subgraph whose nodes consist of statements and predicates is derived from [36, 37], which constructs an operator-level subgraph. This subgraph is most easily built when there are no side-effects due to pointers, shared variables, or procedure calls with other than value parameters. DAGs [1] are constructed for each basic block during the initial parse of the source program. Each upwards exposed use in a block has a corresponding DAG leaf node as usual; these leaves are called *merge nodes*. Data flow analysis is then performed to compute the set of reaching definitions [1, 25, 26] for each basic block, with the additional assumption that every variable is initially assigned the value "undefined" at program entry. Finally, the individual DAGs are connected to one another using the results of the data flow computation: edges are added from definition nodes to the

corresponding merge nodes that may be reached. (This makes definition-use chaining [1] explicit.) A merge node thus represents the *set* of reaching definitions for some variable.

In this construction process, I/O operations are treated as operations on an implicit file object so that the sequencing of operations is correctly represented. Only one node is created for each unique constant to simplify common subexpression elimination. Subscripted array references are represented by a *select* operator having two inputs, an array and an offset, and one output, the selected element. Subscripted array definitions are represented by an *update* operator having three inputs: an array, an offset, and a replacement value. The output of an *update* operator is a modified array. In addition, the definite iteration statement (DO, **for**) becomes a single operator which has as operands the initial, final, and increment values. It has two outputs: one an index value stream and the other a predicate value stream. This operator is essential to our reduction in strength and induction variable substitution schemes [39]: other induction variables are later converted to this form by a separate transformation.

Other edges are necessary for certain transformations. For example, the incremental data dependence update algorithm described in Section 5 requires that *output dependence* [29, 31] edges be inserted during graph construction to force sequencing of multiple definitions of the same object. Output dependence edges are added by treating each definition as a pseudo-use of the defined variable and using the same techniques that create data dependence edges. Vectorization requires that *antidependence* [29, 31] and output dependence edges must be appropriately inserted between array references. Antidependence edges are inserted to force sequencing between a use and a definition of a variable [28, 30].

Dependences between array elements are constructed conservatively as sketched above. As in most optimizing compilers for sequential machines, a definition of an element of an array is considered a definition for the entire array for purposes of data flow analysis and data dependence construction. Similarly, a reference is assumed to be capable of referencing any element of the subscripted array. In order to apply vectorizing transformations, subscript expressions must be analyzed to determine (where possible) accurate dependences. This area has been given a great deal of attention [7, 11, 13, 28, 32, 56]. The following steps are taken to determine array dependences. All loop coindices (induction variables) are located and normalized to run from 1 to N with an increment of 1; operations are "pushed" into the subscript expressions to maintain equivalent values. All subscript expressions are restated in terms of a single induction variable per loop.[8] At this point, the subscript expressions will generally be in the form of a linear function of the loop induction variable. Dependence analysis consists of a pairwise determination of whether the curves for these subscript functions intersect at an integer point within the region defined by the loop induction variable range.

Data dependences can be further classifed as *loop-carried* and *loop-independent* [7, 28, 56]. A loop-carried dependence arises because of the iteration of loops; an example is a definition from one iteration that is referenced on a subsequent

[8] Algorithms for two of the transformations necessary for analysis have been developed for the PDG and are reported elsewhere: induction variable detection [23] and induction variable substitution [39].

iteration. A loop-independent dependence occurs because of execution order, regardless of loop iteration.

3.2.2 *Aliasing, Procedure Calls, and Side Effects.* Aliasing and side-effects present obvious problems in accurately representing dependences in the PDG. All of the known solutions for data flow analysis in the presence of aliasing, pointers, shared variables and procedures [1] are of use here. Explicit aliasing of scalars (e.g., *via* a FORTRAN EQUIVALENCE) is easily handled by treating the aliased names as synonyms. To detect implicit aliasing induced by procedure parameter binding, interprocedural data flow analysis must be performed [4, 12, 14, 15]. This analysis is complicated in FORTRAN, where the passage of array elements can create overlapping alias patterns [20]. Interprocedural analysis is additionally required to detect procedure side-effects (global modifications). Given the information thus obtained, we can represent each procedure call safely: each call is an operator with dependences not only on its parameters, but also on any other shared variables referenced or defined.

Interprocedural analysis must be performed before even the basic block DAGs are constructed. Thus, in the presence of procedures with side-effects, construction of the data dependence subgraph involves translating the source program into triples or some other easily scanned intermediate, performing intra- and interprocedural data flow analysis, and then building the subgraph.

Pointers present the greatest problem. "Well-behaved" pointers, such as those in Pascal, can be analyzed using data flow methods [1]. Pointers in a language such as C can preclude PDG construction altogether since they can point to anything. In the worst case, one would have to conservatively assume that all objects are aliased. Fortunately, it is possible to discover what objects are pointed to when pointer arithmetic is used in a controlled manner. When constraints cannot be computed, interaction with the user might be helfpul in obtaining them.

3.3 Alternative Versions of the PDG

A variety of views of a program may be constructed with variations of the PDG. Nodes can represent statements and predicate expressions, or they can represent individual operators and operands. The latter representation is necessary for most optimizations. This section briefly describes a functional or value oriented view versus an operational or memory mapped view. We also suggest how a program may be represented hierarchically.

A functionally oriented PDG can be constructed that is similar to a data flow graph [18]. The atomic or indivisible nodes of the PDG would be at the operation level without reference to names. In this case, we interpret nodes and edges to simply represent value flow. Arrays would be handled as an entire unit; that is, true dependences between array references would represent the flow of the entire array.

An operationally oriented PDG can be constructed by allowing the assignment operator and references to variable names. In this manner, programs in languages like FORTRAN can be more easily represented. The operational view can also permit memory addressing optimization by allowing a *store* operator that has the following inputs: (1) the memory address to be changed and (2) the replacement

value. An update operator can then be converted to a store: The subscript expression plus the base address of the array becomes the first operand. (An example of the optimization permitted by this representation is reduction in strength [6, 39], whereby a loop index is revised to iterate through *addresses* rather than *subscript values*. In the functionally oriented PDG, reduction in strength could only simplify subscript value computations.)

The PDG can be made hierarchical, that is, its nodes can consist of operations, statements, or higher level groupings if desired. (It is, minimally, hierarchical at the procedure call interface.) Without a hierarchical structure, if a transformation might be predicated on whether some dependence exists within a region, all of the nodes within the region must be visited in order to summarize that information. A hierarchical representation would have summary information for each region posted to each region node during graph creation. Transformations that change individual dependences would then have to also update the summaries for the enclosing regions. Such a hierarchical PDG has been used as the basis for reordering transformations such as vectorization and loop fusion [50]. Its importance lies in its summarization of large sections of the program with respect to both control and data dependence.

3.4 Practical Considerations

The amount of space required to represent a PDG is the dominating practical consideration for this work. We cannot give a useful model for anticipated space consumption since the size of the graph is related to the dependence structure, rather than any easily measured surface feature of a program. At Michigan Tech, Ellcey constructed a FORTRAN-77 front-end that performs interprocedural analysis and builds a PDG [20]. Ellcey's thesis reports the results of translating nine programs containing a total of 40 modules and 2,409 executable statements. He compares the amount of space required for a PDG to that required by a triples representation. The space needed for data flow bit vectors and association (mapping) tables is included in the total triples cost since global optimizations cannot be performed without these additional structures. The ratio of space required for the PDG to space required for triples ranged from 1.13 to 2.18 in the 40 modules in Ellcey's study. It should be noted that it is not clear that the sample used in this study is representative of most programs, but the variance is small. In particular, based on the larger and more realistic programs in the study (two eigenvalue problems and a fluid dynamics problem), we hypothesize that the PDG requires 50 percent more storage than an equivalent triples representation.

4. SOME APPLICATIONS OF THE PDG

This section gives a sample of some of the applications of the PDG. The first four subsections concern optimization questions: the detection of parallelism, node splitting, code motion, and loop fusion. The final subsection gives an application of the PDG in an entirely different context: a software development environment.

All transformations on the PDG are stated as graph walks. As an example of what is involved in such a walk, consider *constant expression folding*, which

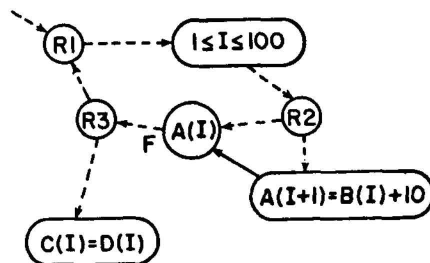

Fig. 8. Representation of array dependences.

```
        DO 100 I = 1, 100
          A(I + 1) = B(I) + 10
          IF (A(I)) GOTO 101
          C(I) = D(I)
100     CONTINUE
101     CONTINUE
```

replaces an operator having constant operands with the result of applying that operator to those operands. One step of constant folding may result in propagating a constant operand to data dependence successors which will then be eligible for constant folding. Thus, given an expression DAG, this optimization should be carried out from the leaves to the root operators. We would express this transformation as a graph walk of the PDG, moving *backwards* from outputs to inputs, constants, or already-visited nodes. As soon as we can proceed no further along a path, we begin descending the traversed path, looking for both expression folding possibilities and other paths to walk up. Thus, no operator is considered for folding until all of its data dependence predecessors (operands) have been considered. In time linear with respect to the number of edges in the PDG, we will have visited every node and performed every possible step of constant expression folding [36]. (This walk is technically a postorder traversal of each spanning tree in the forest obtained by performing a depth first search of the *reverse* PDG, starting at outputs.)

The scalar propagation, common subexpression elimination, and reduction in strength transformations developed for the (extended) data flow graph [23, 36, 37, 39] are also applicable to the PDG and provide further examples of transformations on this kind of program representation.

4.1 Detection of Parallelism

The vectorization algorithms of [41] and [8] operate on the data dependence graph, and vectorize any statement not contained in any strongly connected region of dependences. "If-conversion" must first be performed to represent the control structure of the program as data dependences. This transformation fragments the control structure of the program in such a way that it is difficult to recover the original control structure if analysis discovers that vectorization is not possible. The PDG can be used *directly* to detect parallelism in the code it represents. Consider the program and its corresponding PDG in Figure 8. Any node in the program dependence graph that is *not* contained in a strongly connected region consisting of *both* control and data dependences can be

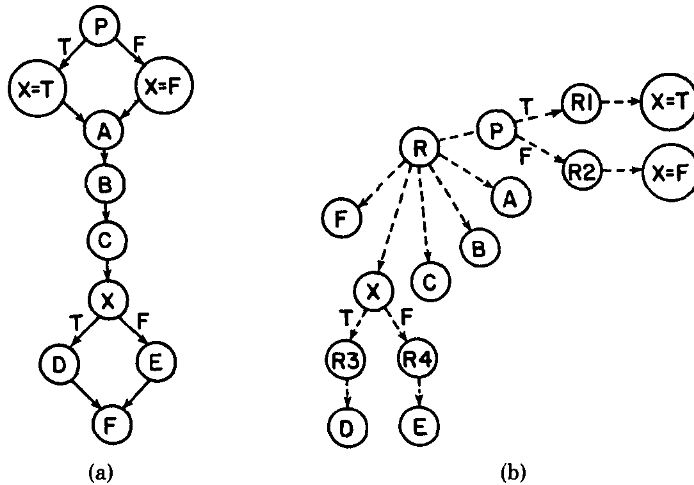

Fig. 9. Candidate for node splitting: (a) control flow graph; (b) control dependence subgraph.

vectorized. The assignment to $A(I + 1)$ *is* part of a strongly connected region and thus is not vectorizable. The assignment to $C(I)$, however, may be vectorized using a WHERE statement [8] following the loop defining A. Thus, the PDG can be used directly as a basis for vectorization, without fragmenting the control structure of the program to the extent determined by "if-conversion". If the (approximate) control dependence subgraph of Section 3.1.2 is used, the quick and easy generation of sequential code is assured if vectorization fails. This latter feature is important in source-to-source transformations, as in [41] and [28].

4.2 Node Splitting

Node Splitting refers to the duplication of a node in a graph and the division of its edges between the two copies to produce an "equivalent" graph [5]. Node splitting has been applied in [31] to break cycles of dependences in the data dependence graph and, hence, generate better code for parallel machines. We believe node splitting also has an important role in process partitioning for multiprocessors in that it can be used to reduce communication and synchronization costs: nodes can be duplicated in several processes to lower communication and synchronization costs between them. The PDG's hierarchical nature allows node splitting to occur at different levels of the hierarchy. In [21], the PDG is used as the basis of several node splitting transformations to eliminate Boolean tests and improve code motion. An example of this application is given below.

An important feature of the PDG that makes its amenable to node splitting is the fact that the execution order of the successors of a region node in the control dependence subgraph is determined entirely by the data dependences between them. In the control flow graph of Figure 9(a), derived from [53], the only definitions of predicate X occur on the branches of P. Hence, if we duplicate nodes X, D, and E on both branches of P, we can eliminate X and thus produce more efficient code. However, to accomplish this we either have to, *in addition*, duplicate the nodes A, B, and C, or determine if X can be "moved up" to A's

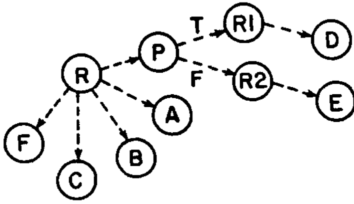

Fig. 10. Optimized code after splitting.

position. In the corresponding control dependence subgraph of the PDG in Figure 9(b), data dependences among successors of R, already present in the PDG, would determine whether X can immediately follow P in execution. Hence no "motion" transformation need be performed with the PDG. If permitted by data dependences, we can immediately split the region determined by node X on both branches of P. Scalar propagation could then eliminate X and dead code elimination would delete the unnecessary copies of $R3$, D, $R4$, and E, obtaining Figure 10. The situation illustrated in Figure 9 occurs naturally in the macro-expansion of languages, where initial parameter settings control subsequent expansion. It also occurs when procedure calls are expanded in-line, a process called *procedure integration* [5].

4.3 Code Motion

The PDG allows powerful transformations, such as invariant code motion, to be performed more efficiently than with other program representations, since dependences in the PDG connect computationally relevant parts of the program. An earlier algorithm permits the safe and efficient motion of large sections of a program, such as loops and conditional blocks, in "structured" programs [23]. That algorithm extends directly to the PDG, permitting such motion in programs with arbitrary control flow. Unlike the methods presented in [1], this extension makes only safe movements, as computations always remain under the influence of the same predicates. In addition, the running time of a program never increases since no code is ever executed in the transformed program unless it would have been executed in the original. As an example of the effects of our code motion algorithm, the PDG for the code fragment in Figure 11(a) would be transformed into a PDG representing the code in Figure 11(b) [23].

Recent work by Cytron, Lowry, and Zadeck [16] extends our code motion algorithm. They present two versions of code motion that incorporate our notion of control dependence. Their strict code motion algorithm is like ours in that the running time of a program is never increased as a result of motion. It represents an improvement over our earlier version in that secondary effects that result from previous code motion are detected without repeated iterations. A nonstrict version, in addition, permits some motion of code dependent on immovable predicates.

We now sketch how the Cytron–Lowry–Zadeck algorithms can be applied to the PDG. A renaming transformation is first performed on a three-address intermediate representation; the necessary renaming can also be represented in the PDG. Birthpoints [42] for variables are then introduced; these birthpoints correspond to merge nodes in the PDG after common subexpression elimination

```
read (a, b, c)
while a < b do
  d = 0
  repeat
    if Q then e = c + 1
         else e = c + d
    d = d + b * c
  until d > c
  a = a * d + b/c
endwhile
```

(a)

```
read (a, b, c)
if a < b then
  t1 = not Q
  d = 0
  t2 = b * c
  if Q then e = c + 1

  repeat
    if t1 then e = c + d
    d = d + t2
  until d > c
  t3 = b/c
  repeat
    a = a * d + t3
  until a >= b
endif
```

(b)

Fig. 11. Example of code motion.

has been applied to them. The algorithms then operate on an interval-by-interval basis, visiting each basic block in an interval in dominated topological order, and moving code from inner to outer intervals one level at a time. As described, this interval structure is directly represented in a hierarchical manner in the PDG. By using the same order within an interval, and summarizing results on an interval by interval basis, we can apply their algorithms directly to the PDG, obtaining an efficient and powerful code motion strategy.

The effectiveness of code motion can be increased by performing unswitching [5], loop splitting [40], and loop peeling.[9] These transformations create more opportunities for the motion of invariants. The following four transformations may be incorporated into the motion strategy, operating on loops in an innermost to outermost order. Here an *exit predicate* is one that controls an exit out of a loop and a *branch predicate* is any other predicate.

(1) Eliminate (partially) invariant branch predicates by unswitching.
(2) Peel a loop with an invariant exit predicate, when that removes an exit from the loop.
(3) Split loops with branch predicates dependent only on the index variable, thereby removing the branches.
(4) Delete an invariant assignment from a loop by peeling off one iteration.

The criterion used to achieve these effects is the absence of data dependence predecessors for a node inside the loop region. Transformation 4 above is, in addition, signaled by a backwards true dependence from the assignment to a use. As an example of 1 above, consider Figure 12(a) where A is an invariant. The

[9] An example of *loop unswitching* is given in Figure 12. *Loop splitting* refers to copying the loop and splitting the original iterations between the two copies (preserving the iteration order). *Loop peeling* is taking the first iteration out of the loop, and adjusting the loop accordingly.

```
      DO 100 I = 1, 100
        IF (A.OR.B(I))C(I) = 0
        D(I) = 0
100   CONTINUE
```

(a)

```
      IF (A) THEN
        DO 100 I = 1, 100
          C(I) = 0
          D(I) = 0
100     CONTINUE
      ELSE
        DO 115 I = 1, 100
          IF (B(I)) C(I) = 0
          D(I) = 0
115       CONTINUE
      ENDIF
```

(b)

Fig. 12. Example of loop unswitching.

"or" operation will have exactly two data predecessors, one of which, A, is outside the loop. If the loop is unswitched with A as the outer predicate, Figure 12(b) is the result. Transformations of one type, such as unswitching, may uncover opportunities for transformations of another type. As a result, all invariant operations are moved to the outermost nesting level possible. An additional extension to the invariant motion algorithm that involves node splitting is outlined in [21].

4.4 Loop Fusion

The hierarchical PDG is the basis for efficient algorithms for many reordering transformations. Loop fusion [5] is a transformation in which the bodies of two loops are joined or fused to create a single loop. This transformation is used to create sectioned code for vector machines as well as to reduce loop overhead in a general optimization setting. In this section, we show that the hierarchical nature of the PDG allows the conditions under which fusion can be performed to be easily checked.

Conditions that permit loop fusion have been formulated in [1], [5], and [50], among others. The simplest case, as formulated in [5], is as follows:

(1) the loops are executed under exactly the same control conditions,
(2) there is no data dependence between the two loops, and
(3) the loops are executed the same number of times.

The first condition is easily and safely approximated[10] in the PDG by checking that both loops have the same control dependence predecessor. Likewise, the second condition involves checking that there is no data dependence edge between the nodes representing the two loops in the hierarchical PDG. The third condition requires checking that the two loops have exactly the same loop control predicates. In the case of definite loops, such as DO loops, the number of times the loops iterate must be the same. Given that, in practice, there are a bounded number of exits from a loop, whether to fuse two loops can be decided by a

[10] The precise problems stated in conditions 1, 2, and 3 are unsolvable.

constant-time check of the hierarchical PDG. In [1], condition (2) is generalized to (in effect) require that no loop-carried (Section 3.2.1) dependence exists between the two loops. This condition is again easily checked in the hierarchical PDG. The more general loop fusion algorithm of [50] can be directly applied to the hierarchical PDG since it is formulated in terms of data dependence.

4.5 Slicing

The PDG is also useful in contexts other than optimization. One such context is in performing slicing in a software development environment. *Slicing* is the abstraction of sets of statements that influence the value of a variable at a particular program location [54, 55]. An experiment by Weiser [55] indicated that programmers use slices when debugging and could therefore benefit from the development of a tool to provide slicing information automatically. A potential software development environment using slicing could allow the user to edit a program using slice membership as the basis for inclusion in a window rather than syntactic structure alone. A debugger could display the offending slice in a window on an error condition or breakpoint rather than the entire syntactic context. (We assume a display routine that would provide automatic eliding, as is done in LISPEDIT [2, 35]. Partial slices, without elision, are displayed for COBOL programs by [47].)

The extraction of slices is based on data dependence; however, control dependence is considered in the construction of a slice as well. A computation which affects the value of a variable at a desired observation point may be under the influence of a predicate (i.e., executed only when a predicate has a particular truth value). Those statements that make up the control structure using the predicate must be included in the slice. This is the key to why the PDG is so appropriate for this technique. A slice is directly obtained by a linear time walk backwards from some point in the graph, visiting all predecessors. (Nodes must be annotated with pointers to the source code in order to permit the display of the obtained slice.) These slices [38] are more accurate than those obtained with earlier methods [54].

5. INCREMENTAL DATA FLOW UPDATE AND INCREMENTAL OPTIMIZATION

The update of data flow information to reflect changes in control flow has been a stumbling block for optimizing compilers. The expense of a full data flow analysis may be avoided if an *incremental* method [43, 53] can be used to examine and update *only* the affected data dependences. An incremental update in the context of a compiler in the process of transforming a given source program is much simpler than the problem of keeping data flow information correct after each editing step in an interactive program development environment. This is because arbitrary changes are permitted in the latter case, but only very specific changes are permitted in the former. In this paper, we do not address the more difficult problem posed by arbitrary changes. In the approach presented here, data dependences are updated as part of certain optimizations. During this update, we can apply *additional* optimizations to the precise program region for

which a profit will result. We call this concept *incremental optimization* to contrast it with conventional optimizations that are designed as passes that operate on an entire module. Other incremental flow algorithms operate on information disjoint from the intermediate program representation and thus cannot guide incremental optimization except on larger regions than may be necessary.

This section describes a new incremental data flow analysis algorithm that operates *directly* on the PDG. We will describe its operation in the context of two optimizations that alter the control flow of a program: *branch deletion* and *loop peeling*. *Branch deletion* kills blocks of code determined to be unreachable due to a conditional branch that will always select the same path. The value of the branch predicate may be determined immediately, or after applications of transformations such as scalar propagation, procedure integration, and constant expression folding. *Loop peeling* moves one or more initial iterations of a loop out of the loop into a header region. This frequently decreases the number of dependences within the remaining loop.

5.1 Branch Deletion

The control dependence update in the PDG is easily performed when a branch is deleted. Suppose a predicate node P is found to be always **true**. If there is a P-*false* region node, it is pruned along with all of its control dependence successors. If there is a P-*true* region node, all successors of the region node are made successors of P's control dependence predecessor and then both P and the region node are deleted.

The data dependence update focuses on definitions formerly transmitted by the deleted control flow path. It is trivial to eliminate data dependences due to definitions on the killed path: The edges are simply deleted when the killed region node is pruned. The nontrivial problem is when the formerly transmitted definition is not *on* the path being deleted, but reaches a use *via* the path. If the deleted path is the only one that allows a definition to reach a use, the corresponding data dependence must be eliminated. When there are *several* definition-free paths from a definition to a use, deletion of one such path has no effect. (Note that it is not possible, by branch deletion, for new definitions to reach uses. Thus, the update question is simply "Which def-use chains must be deleted?".)

Example. Let us first consider an example of the general data flow issue before considering the update solution with the PDG. Suppose that we have the fragment below:

$B =$	(1)
if P **then**	(2)
$A =$	(3)
$B =$	(4)
else	(5)
$A =$	(6)
endif	(7)
$= A$	(8)
$= B$	(9)

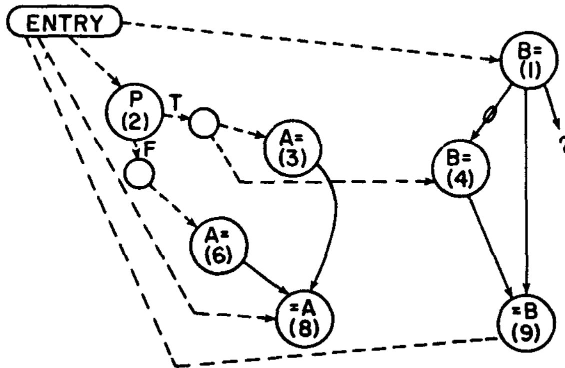

Fig. 13. PDG for branch deletion.

Now, suppose that it is determined that the predicate expression P is *always* **true**. Branch deletion will remove the test and the **else** clause of the **if** (lines 2 and 5–7). Consider the effect in terms of the reaching definitions at lines 8 and 9. The **else** clause contained a definition of A that should simply be removed from the set of reaching definitions for line 8. The **else** clause *also* provided a definition-free path along which definition B at line 1 reached line 9. With the deletion of this path, $B1$[11] no longer reaches line 9 as it is killed by $B4$. The data flow update should thus show reaching definitions of $A3$ and $B4$ for lines 8 and 9. With this updated information, additional optimizations might now be possible: Subsumption or scalar propagation on lines 8 and 9 could offer benefits and it is possible that definition $B1$ can be deleted if line 9 was its only use.

We now return to the PDG and develop in stages an incremental solution to this update problem that examines only the affected data dependences. Consider the PDG for the previous example, shown in Figure 13. (The nodes are parenthetically numbered with the corresponding source line number for reference purposes only. A solid edge with the letter "O" on it represents an *output dependence* edge; see Section 3.2.1.) Assume that we have just determined that P is always **true**. Everything in the *P-false* region must be deleted. Here, node (6) and all edges incident to it and exiting from it are deleted. The predicate computation node is now deleted. Anything that had been in a *P-true* region is now placed in a region dependent only on P's former control dependence predecessor ("entry") by simply attaching the *P-true* control dependence edges to that predecessor. The control dependence update is now complete, as is part of the data dependence update.

To complete the update of affected data dependences, we examine all of the nodes that were in the old *P-true* region. Let d be such a node. Consider all pairs (d, p) where d is *output dependent* upon predecessor p, and d and p have the same node m as a data dependence successor. There are two cases on (d, p) to consider. The first case examines the least common ancestor of d and p, denoted by $\text{LCA}(d, p)$.[12] A data flow anomaly exists if $\text{LCA}(d, p)$ equals $\text{CP}(d)$ since this

[11] We denote a definition of a variable V on line i as Vi.

[12] We use $\text{CP}(d)$ to denote the control dependence predecessor of d and $\text{LCA}(d, p)$ to denote the least common control ancestor of d and p in a depth first spanning tree of the predicate graph.

relationship would imply that p could reach m in spite of the fact that any path to m must go through d. Consider:

```
if P then A =      (1)
if Q then A =      (2)
  = A              (3)
```

If Q is found to be **true**, A2 (d) should then kill A1 (p). This suggests the following transformation.

Case 1. If CP(d) = LCA(d, p), d should prevent p from reaching m. Delete the edge from p to m. If p has no other data dependence successors, delete it as a useless definition. (Continue this dead code pruning transitively.)

In Figure 13, only node pair (4, 1) with successor (9) satisfies the above criteria after deletion of P since CP(4) = CP(1) = LCA(4, 1) = "entry". The edge (1, 9) should be deleted. We can now perform *incremental optimization* based on this update. Node 1 should be examined for dead code removal. As node 9 now has only one reaching definition, it is reasonable to perform scalar propagation on it, as well as other transformations such as common subexpression elimination.

Now, consider all pairs (d, s) where s is an *output dependent* successor of d, d and s have the same node m as a data dependence successor, and, as above, d is in the old *P-true* region. Here, an anomaly exists if CP(d) = LCA(d, s) and (d, m) is loop-independent while (s, m) is loop-carried.[13] As an example, consider:

```
loop                (1)
  if P then A =     (2)
  = A               (3)
  A =               (4)
endloop             (5)
```

where A2(d) kills A4(s) for the use of A at line 3 if P is found to be **true**.

Case 2. If CP(d) = LCA(d, s) and (d, m) is a *loop-independent* data dependence edge while (s, m) is a *loop-carried* data dependence edge, then d should prevent s from reaching m. The transformation on s is the same as that on p above.

These two restricted cases illustrate the update task, but do not solve the general problem. Consider Figure 14, a simple example of the general case. Note that if P is found to be always **true**, definition $A1$ can no longer reach the use of A. It is killed on both remaining paths. In cases (1) and (2), a single definition killed another. In the general case, several definitions, along several paths, may kill a definition. The problem can be solved by transitively abstracting the definitions in a region to a predicate or region node. We create a *pseudo-definition* for a variable in a region, and a corresponding dependence edge if *all* paths in the region have a definition of that variable with that dependence.[14] Our previous transformations can then be applied. In Figure 14, definitions $A3$ and $A4$ would be abstracted to a pseudo-definition $A2$ representing the entire conditional region. This abstraction occurs because *both* paths from (2) to (5)

[13] See Section 3.2.1 for a definition of loop-independent and loop-carried dependences.

[14] This differs from the dependences abstracted for the hierarchical representation suggested in Section 3.3. There, a dependence exists if *any* path in the summarized region has such a dependence.

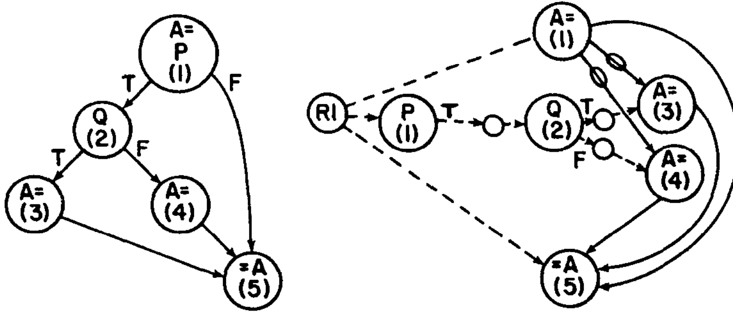

Fig. 14. Schema with multiple definition-free paths.

contain definitions of A. The pseudo-definition $A2$ would be output dependent on $A1$, just as $A3$ and $A4$ are. Case 1 above could then produce the correct update given this pseudo-definition.

Psuedo-definitions are posted to region nodes if *some* successor has a definition. This is because a region represents a basic block in the conventional sense: all operators are evaluated if any one is. Predicates are annotated with pseudo-definitions when there are definitions with the same output dependence on *both* branches. These pseudo-definitions are posted as long as there is information to propagate. Pseudo-definition propagation can continue as far as the outermost node that is the least common ancestor of a definition in the altered *P-true* region and another definition p upon which it is output dependent. During this propagation, the only definitions that need be examined are those that are also output dependent on p.

The complexity of this incremental update has, of course, a worst-case performance equivalent to reanalyzing the entire program. We do not expect this worst case to occur and refer the reader to [43] for a good discussion of complexity analysis for incremental flow methods. Only experimentation will demonstrate how this method may perform in practice. We feel that it will be beneficial to perform the "Case 1" and "Case 2" transformations first to prune a region before the region abstraction walk occurs. No wasted work is performed in these fast cases and any pruning can only save time during abstraction.

5.2 Loop Peeling and Unrolling

Loop peeling involves duplicating the code in the body of a loop. When one iteration is peeled, this code is tailored for the initial loop index values and the loop control is revised to make one less iteration. There may be dependences on computations external to the loop that should now belong solely to the peeled iteration and that should be deleted from the remaining loop. Consider the loop in Figure 15(a), which is transformed into a header and a new loop in Figure 15(b). A control dependence due to the label L has been removed from the loop, permitting better pipelining and simplifying the control dependence structure. In addition, the reference to A within the loop is now only dependent on the definition of A in the loop. In the original loop, that reference also includes external definitions of A since the loop could be entered *via* label L.

```
                              if P then
                                A =
                          L:    B = A ...
                              endif
      while P do              while P do
        A =                     A =
L:      B = A ...               B = A ...
      endwhile                endwhile
        (a)                     (b)
```

Fig. 15. Loop peeling: (a) original loop; (b) one iteration peeled.

To construct the dependences for the duplicated code, output dependence edges must be added from the final definition nodes in the peeled header (the definitions "available on exit" from the header) to the initial corresponding nodes in the loop (the "exposed uses"). The remaining loop body must have its control dependences reduced to be only the loop predicate, all alternate-entry control dependences belonging to the header. Then, the data dependences within the loop body must be updated. (In the case of a definite loop, e.g., a DO or **for**, the header iteration should be optimized for the initial index values. Other optimizations possible include invariant code motion. As the header dominates the loop body, common subexpression elimination gives us the equivalent of invariant code motion on the original loop.)

The incremental dependence update for the reference to A within the loop requires a new condition to recognize the anomaly introduced by peeling away an iteration that contains a label. Like Case 1, we consider all pairs (d, p) where d is output dependent upon predecessor p and both d and p have the same node m as a data dependence successor. (In the example at hand, the definitions of A are d and p and the reference to A is m.)

Case 3. If (d, m) is *loop-independent*, d should prevent p from reaching m. The transformation on p should be as in Case 1.

A transformation that is closely related to loop peeling is *loop unrolling*. There, the code within the loop may be duplicated one or more times and the number of loop iterations decreased proportionately, in order to permit better loop body pipelining and decrease the overhead of branch execution and induction variable modification. The actions required to update dependences when performing unrolling are so similar to those involved in peeling that we do not discuss them here.

6. SUMMARY AND FUTURE WORK

A new program representation, called the *program dependence graph* or *PDG*, has been presented and shown to permit efficient and powerful program transformations. By representing data and control dependences explicitly, the PDG provides a unifying framework in which previous work in compiler optimization can be viewed since most optimizing transformations are in essence operating on a program's dependence structure. Examples of vectorization, node splitting, code motion, and loop fusion were given. An incremental data flow update algorithm was presented that permits incremental optimization as the update progresses.

The importance of the PDG lies in the fact that potential parallelism in the program is exposed since artificial orderings are eliminated in favor of only the data and control dependence structure. We expect to use the PDG as the basis for process partitioning for multiprocessors. The hierarchical PDG lends itself to a top-down partitioning of a program into processes; in addition, the PDG provides a promising framework for clustering operations and statements into processes in a bottom-up approach.

APPENDIX. Graph Theoretic Terminology

A *directed graph* $G = (N, E)$ is a finite set of *nodes* N together with a set E of ordered pairs of nodes called *edges*. If (n, m) is an edge, then n is a *predecessor* of m and m is a *successor* of n. A graph $G' = (N', E')$ is a *subgraph* of G if N' is contained in N and E' is contained in E. A *path* from n to m in G is a finite sequence of nodes $v_0 = n, v_1, \ldots, v_k = m$ such that (v_i, v_{i+1}) is in E for $0 \leq i < k - 1$. The *reverse graph* of a graph $G = (N, E)$ is a graph (N, E'), where E' consists of edges (n, m) such that (m, n) is in E. A *strongly connected region* of a graph G is a subgraph of G in which there is a path from every node to every other node.

ACKNOWLEDGMENTS

We thank Fran Allen for having posed many of the problems solved in this paper, and Michael Burke, Larry Carter, Linda Ottenstein, and the referees for their helpful comments. Andy Lowry substantially simplified our control dependence generation algorithm by noting how the computation of least common ancestors could be avoided. Ron Cytron and Kenny Zadeck helped clarify the differences between their code motion algorithm and ours. The support of the IBM Corporation for Joe Warren as a 1983 summer student and for Karl Ottenstein as a consultant at IBM Yorktown is also appreciated.

REFERENCES

1. AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986. [An alternative reference is this book's predecessor, AHO, A. V. AND ULLMAN, J. D. *Principles of Compiler Design.* Addison-Wesley, 1977.]
2. ALBERGA, C. N., BROWN, A. L., LEEMAN, G. B., JR., MIKELSONS, M., AND WEGMAN, M. N. A program development tool. *IBM J. Res. Dev 28*, 1 (Jan. 1984).
3. ALLEN, F. E. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization. SIGPLAN Not. 5*, 7 (July 1970), 1–19.
4. ALLEN, F. E. Interprocedural analysis and the information derived by it. In *Lecture Notes in Computer Science Vol. 23*, Springer-Verlag, New York, 1974, 291–321.
5. ALLEN, F. E., AND COCKE, J. A catalogue of optimizing transformations. In *Design and Optimization of Compilers*, Randall Rustin, Ed., Prentice-Hall, Englewood Cliffs, NJ, 1972, 1–30.
6. ALLEN, F. E., COCKE, J., AND KENNEDY, K. Reduction of operator strength. In *Program Flow Analysis, Theory and Applications*, S. Muchnick and N. Jones, Eds., Prentice-Hall, Englewood Cliffs, NJ, 1981, 79–101.
7. ALLEN, J. R. Dependence analysis for subscripted variables and its application to program transformations. Ph.D. dissertation, Dept. of Computer Science, Rice University, Houston, TX, April 1983.
8. ALLEN, J. R., KENNEDY, K., PORTERFIELD, C., AND WARREN, J. Conversion of control dependence to data dependence. In *Conference Record of the 10th Annual ACM Symposium on Principles of Programming Languages* (Austin, Texas, Jan. 24–26, 1983), ACM, New York, 177–189.

9. ALLEN, J. R., AND KENNEDY, K. PFC: a program to convert FORTRAN to parallel form. Tech. Rep. 82-6, Dept. of Mathematical Sciences, Rice University, March 1982, 63 pages.
10. ALLEN, R., AND KENNEDY, K. Programming environments for supercomputers. In *Supercomputers: Algorithms, Architectures, and Scientific Computation*, F. A. Matsen and T. Tajima, Eds., University of Texas Press, Austin, TX, 1986, 19–38.
11. BANERJEE, U. Data dependence in ordinary programs. Report 76-837, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, Nov. 1976.
12. BURKE, M. An interval analysis approach toward interprocedural data flow analysis. IBM Research Report RC 10640, T. J. Watson Research Center, Yorktown Heights, NY, July 1984.
13. BURKE, M., AND CYTRON, R. Interprocedural dependence analysis and parallelization. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction* (Palo Alto, CA, June 25–27, 1986). *ACM SIGPLAN Not. 21*, 7 (July 1986), 162–175.
14. COOPER, K., AND KENNEDY, K. Efficient computation of flow insensitive summary information. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction* (Montreal, June 17–22, 1984). *ACM SIGPLAN Not. 19*, 6 (June 1984), 247–258.
15. COOPER, K. Analyzing aliases of reference formal parameters. In *Conference Record of the 12th ACM Symposium on Principles of Programming Languages* (New Orleans, LA, Jan. 14–16, 1985), ACM, New York, 281–290.
16. CYTRON, R., LOWRY, A., AND ZADECK, K. Code motion of control structures in high-level languages. In *Conference Record 13th Annual ACM Symposium on Principles of Programming Languages* (St. Petersburg, FL, Jan. 13–15, 1986), ACM, New York, 70–85.
17. DAVID, A. L., AND KELLER, R. M. Data flow program graphs. *IEEE Comput. 15*, 2 (Feb. 1982), 26–41.
18. DENNIS, J. B. First version of a data flow procedure language, revised Comp. Struc. Group Memo 93 (MAC Tech. Memo 61) MIT LCS (May 1975) 21 pages.
19. DENNIS, J. B. Data flow supercomputers. *IEEE Comput. 13*, 11 (Nov. 1980), 48–56.
20. ELLCEY, S. J. The program dependence graph: interprocedural information representation and general space requirements. Master's thesis, Dept. of Computer Science, Michigan Technological Univ., Houghton, MI, Aug. 1985.
21. FERRANTE, J. The program dependence graph as a basis for node splitting transformations. IBM Research Rep. RC 10542, Yorktown Heights, NY, June 1984.
22. FERRANTE, J., AND MACE, M. On linearizing parallel code. In *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages* (New Orleans, LA, Jan. 14–16, 1985), ACM, New York, 179–190.
23. FERRANTE, J., AND OTTENSTEIN, K. A program form based on data dependency in predicate regions. In *Conference Record of the 10th Annual ACM Symposium on Principles of Programming Languages* (Austin, TX, Jan. 24–26, 1983), ACM, New York, 217–231.
24. FERRANTE, J., OTTENSTEIN, K., AND WARREN, J. D. The program dependence graph and its use in optimization. *Lecture Notes in Computer Science Vol. 167*, Springer-Verlag, 1984, 125–132.
25. GRAHAM, S., AND WEGMAN, M. A fast and usually linear algorithm for global flow analysis. *J. ACM 23*, 1 (Jan. 1976), 172–202.
26. HECHT, M. S. *Flow Analysis of Computer Programs*. Elsevier-North Holland, New York, 1977.
27. KAS'JANOV, V. N. Distinguishing hammocks in a directed graph. *Soviet Math. Doklady 16*, 5 (1975), 448–450.
28. KENNEDY, K. Automatic translation of FORTRAN programs to vector form. Report 476-029-4, Dept. of Mathematical Sciences, Rice Univ., Houston, TX, June 1980.
29. KUCK, D. J. *The Structure of Computers and Computations*. Wiley, New York, 1978.
30. KUCK, D. J., KUHN, R. H., LEASURE, B., AND WOLFE, M. The structure of an advanced vectorizer for pipelined processors. In *Proceedings IEEE 4th International COMPSAC* (Chicago, IL, Oct. 1980), IEEE, New York, 709–715.
31. KUCK, D. J., KUHN, R. H., PADUA, D. A., LEASURE, B., AND WOLFE, M. Dependence graphs and compiler optimizations. In *8th Annual ACM Symposium on Principles of Programming Languages* (Williamsburg, VA, Jan. 26–28, 1981), ACM, New York, 207–218.
32. KUHN, R. H. Optimization and interconnection complexity for parallel processors, single-stage networks and decision trees. Ph.D. dissertation, Rep. 80-1009, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, Feb. 1980.

33. LENGAUER, T., AND TARJAN, R. E. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst. 1*, 1 (July 1979), 121–141.
34. MCGRAW, J., SKEDZIELEWSKI, S., ALLAN, S., GRIT, D., OLDEHOEFT, R., GLAUERT, J., DOBES, I., AND HOHENSEE, P. SISAL: streams and iteration in a single-assignment language. Language reference manual version 1.1. Report M-146, Lawrence Livermore National Laboratory, July 20, 1983.
35. MIKELSONS, M. Prettyprinting in an interactive environment. In *Proceedings of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation* (Portland, OR, June 1981), ACM, New York, 108–116.
36. OTTENSTEIN, K. J. Data-flow graphs as an intermediate program form. Ph.D. dissertation, Computer Sciences Dept., Purdue University, Lafayette, IN, Aug. 1978.
37. OTTENSTEIN, K. J. An intermediate program form based on a cyclic data-dependence graph. CS-TR 81-1, Dept. of Computer Science, Michigan Technological Univ., Houghton, MI, Oct. 1981; July 1982, errata.
38. OTTENSTEIN, K. J., AND OTTENSTEIN, L. M. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGPLAN/SIGSOFT Symposium on Practical Programming Development Environments* (Pittsburgh, PA, April 23–25, 1984), *ACM SIGPLAN Not. 19*, 5 (May 1984), 177–184, and *Softw. Eng. Notes 9*, 3.
39. OTTENSTEIN, K. J. A simplified view of reduction in strength. CS-TR 85-4c, Michigan Technological Univ., Houghton, MI, Aug. 1986.
40. PADUA, D. A. Multiprocessors: discussion of some theoretical and practical problems. Ph.D. dissertation, Computer Sciences Dept., Univ. of Illinois, Urbana-Champaign, Sept. 1979.
41. PADUA, D. A., KUCK, D. J., AND LAWRIE, D. High-speed multiprocessors and their compilers. *IEEE Trans. Comput. 29*, 9 (Sept. 1980), 763–776.
42. REIF, J. H., AND LEWIS, H. R. Symbolic evaluation and the global value graph. In *Conference Record of the 4th Annual ACM Symposium on Principles of Programming Languages* (Los Angeles, CA, Jan. 17–19, 1977), ACM, New York, 104–118.
43. RYDER, B. G. Incremental data flow analysis. In *Conference Record 10th Annual ACM Symposium on Principles of Programming Languages* (Austin, TX, Jan. 1983), ACM, New York, 167–176.
44. SARKAR, V., AND HENNESSY, J. Compile-time partitioning and scheduling of parallel programs. In *Proceedings of the ACM SIGPLAN Compiler Construction Conference* (Palo Alto, CA, June 25–27, 1986), *ACM SIGPLAN Not. 21*, 7 (July 1986).
45. SARKAR, V., AND HENNESSY, J. Partitioning parallel programs for macro-dataflow. In *Proceedings of the ACM Conference on LISP and Functional Programming* (Cambridge, UK, Aug. 4–6, 1986), ACM, New York, 202–211.
46. SKEDZIELEWSKI, S., AND GLAUERT, J. IFl: an intermediate form for applicative langauges, draft 4. Lawrence Livermore National Laboratory, Livermore, CA, June 26, 1983.
47. TISCHLER, R., SCHAUFLER, R., AND PAYNE, C. Static analysis of programs as an aid to debugging. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging* (Pacific Grove, CA, March 20–23, 1983), *ACM Softw. Eng. Notes 8*, 4 (Aug. 1983) and in *ACM SIGPLAN Not. 18*, 8 (Aug. 1983), 155–158.
48. TOWLE, R. A. Control and data dependence for program transformations. Ph.D. dissertation, Univ. of Illinois, Urbana-Champaign, Feb. 1976.
49. TRELEAVEN, P. C., HOPKINS, R. P., AND RAUTENBACH, P. W. Combining data flow and control flow computing. *Comput. J. 25*, 2 (1982), 208–217.
50. WARREN, J. A hierarchical basis for reordering transformations. In *Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages* (Salt Lake City, UT, Jan. 1984), ACM, New York, 272–282.
51. WATERS, R. C. The programmer's apprentice: knowledge based program editing. *IEEE Trans. Softw. Eng. SE-8*, 1 (Jan. 1982), 1–12.
52. WATERS, R. C. Automatic analysis of the logical structure of programs. AI-Lab TR-492, MIT, Cambridge, MA., Dec. 1978. Available as NTIS AD-A084 818.
53. WEGMAN, M. Summarizing graphs by regular expressions. In *Conference Record of the 10th Annual ACM Symposium on Principles of Programming Languages* (Austin, TX, Jan. 24–26, 1983), ACM, New York, 203–212.

54. WEISER, M. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering* (San Diego, CA, March 9–12, 1981), IEEE Computer Society Press, New York, 439–449.
55. WEISER, M. Programmers use slices when debugging. *Commun. ACM 25*, 7 (July 1982), 446–452.
56. WOLFE, M. J. Optimizing supercompilers for supercomputers. Ph.D. dissertation. Tech. Rep. UIUCDCS-R-82-1105, Univ. of Illinois, Urbana-Champaign, Oct. 1982.

Received June 1984; revised August 1986; accepted October 1986