The Combining DAG: A Technique for Parallel Data Flow Analysis

Robert Kramer, Rajiv Gupta, and Mary Lou Soffa

***Abstract*— As the number of available multiprocessors increases, so does the importance of providing software support for these systems, including parallel compilers. Data flow analysis, an important component of software tools, may be computed many times during the compilation of a program, especially when compiling for a multiprocessor. Although converting a sequential data flow algorithm to a parallel algorithm can present some opportunities for computing data flow in parallel, more parallelism can be exposed by the development of new parallel data flow algorithms. In this paper, we present a technique that computes rapid data flow problems in parallel and thus is applicable for commonly used classical data flow problems, including reaching definitions, reachable uses, available expressions, and very busy expressions. Unlike previous techniques, our technique exploits the inherent parallelism in the data flow computation that occurs across independent paths, within linear paths, and in paths through loops of a control flow graph. The technique first changes cyclic structures in a control flow graph to acyclic structures and then builds a combining directed acyclic graph (DAG) that represents the paths through the control flow graph needed to compute data flow. Data flow is then computed using two passes over the DAG by computing the data flow for the nodes on each level of the DAG in parallel. We also present experimental results comparing the performance of our algorithm with a sequential algorithm and a parallelized sequential algorithm.**

***Index Terms*—Data flow analysis, control flow graph, iterative method, structured analysis, node listings**

I. Introduction

ALTHOUGH parallelizing compilers continue to receive much research attention, the increased availability of parallel processors has recently generated an increased interest in the design and implementation of parallel compilers [6], [17]. With the availability of multiprocessor environments, we no longer need to use a uniprocessor to compile programs that are targeted for a multiprocessor system, but rather can use the multiprocessor itself. A very important component of any compiler is data flow analysis. Data flow information is used in various phases of compilation, including code generation, code scheduling, traditional optimizing transformations, and parallelizing transformations. Consequently, a single program compilation may involve the computation of the solution to several data flow problems at different points in the language translation process. For large programs, the data flow analysis performed during compilation can be time-consuming. Also, a number of important code transformations, such as loop unrolling, loop splitting, and in-line code substitution, increase the code size. This larger code size further adds to the cost of data flow analysis. Finally, data flow analysis is also used in other program tools, such as editors, testers, and debuggers, in both sequential and parallel programming environments [2]–[5], [8], [14], [15]. The computation of data flow in parallel is beneficial for a number of tools running in a multiprocessor environment.

In developing parallel algorithms, there are two approaches that can be taken. In one approach, a sequential algorithm is adapted to a parallel version of the algorithm. Using this approach, moderate amounts of parallelism can typically be detected. The other approach is to develop novel techniques and algorithms to solve a problem that more fully exploit the inherent parallelism of the problem. In this way, more parallelism is exposed. Both approaches have been used in recent algorithms for parallelizing data flow analysis [7], [13], [18].

To compute data flow information in parallel, the data flow problem is decomposed into subproblems, which are solved in parallel. The results from the subproblems are then combined to obtain a data flow solution for the entire program. The decomposition is achieved by partitioning the control flow graph and decomposing the data flow problem into subproblems associated with these partitions. When adapting a sequential algorithm to a parallel algorithm, the partitioning of the data flow problem is based upon the natural partitioning imposed by the data flow analysis algorithm, such as intervals or maximal strongly connected components. Parallel versions of the sequential Allen–Cocke interval analysis algorithm [1], [7], [18] and a parallel version of a hybrid method for computing data flow [12], [13] have been developed using the natural partitioning of the sequential algorithms.

In the parallel interval analysis algorithm, computation for each interval in a given graph of a derived sequence is performed in parallel [7], [18]. A similar approach uses a hybrid algorithm [13] that combines an iterative technique with a structured technique to compute data flow [12]. In this parallel algorithm, the maximal strongly connected components of a program are detected, and local data flow problems are solved, for each component in parallel. A propagation phase distributes the data flow information to the components. Global data flow is then computed for the blocks in each component in parallel. The major problem with these algorithms is that there is no control over the decomposition of the

Manuscript received June 16, 1992; revised June 16, 1993. This work was supported in part by the National Science Foundation under Presidential Young Investigator Award CCR-9157371 and under Grant CCR-9109089 to the University of Pittsburgh.

The authors are with the Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260 USA; e-mail: gupta@cs.pitt.edu.

IEEE Log Number 9401218.

flow graph. It could happen that the sizes of the intervals or strongly connected components (i.e., loops) vary considerably. It could also happen that the flow graph decomposes into very few partitions. For example, if a program had one major loop, then there would be only one partition, and no parallelism would be exploited. Both the size and the number of partitions are determined purely by the structure of the program and the natural partitioning of the algorithm. Thus, some components could be very large, whereas others are not large at all, causing problems in mapping to the processors. The dependence on the program structure restricts the exploitation of the parallelism in the computation of data flow.

The second approach of developing parallel algorithms has been used in the development of techniques that partitions a control flow graph without consideration of the flow graph structure [7]. In this paper, we present a new technique for computing parallel data flow that is applicable to structured control flow graphs and solves rapid data flow problems [9], [10]. Thus, it is oriented toward the four classic (and commonly used) data flow problems, namely, reaching definitions, live uses of variables, available expressions, and very busy expressions. The technique detects the inherent parallelism in data flow analysis using three sources.

1) The parallelism in the data flow computation across independent paths (traces) in a control flow graph is detected.
2) Within each independent path, the parallelism in computing data flow for a linear path is detected by path partitioning.
3) The parallelism in data flow paths through loops in the control flow graph is detected by transforming the cyclic structures to acyclic structures. The resulting acyclic structure enables the computation of data flow information within the loop to be carried out in parallel with the computation of data flow information for code preceding the loop and with code following the loop (overlapped execution).

In our approach, all three possible parallel opportunities are detected. Using the control flow graph, the technique first converts the cyclic structures of a control flow graph into acyclic structures, and then constructs a directed acyclic graph (DAG) from the acyclic control flow graph, in which each node represents subpaths in the control flow graph. The data flow for subpaths is computed and combined with other subpaths. The data flow for nodes at each level in the DAG is computed in parallel. The decision about the extent to which this parallelism can be exploited will be made based upon the communication costs on a specific parallel architecture. Depending on the architecture, the nodes of the DAG can either be scheduled to execute in parallel or be merged with other nodes and scheduled using standard scheduling techniques, such as proposed by Sarkar [16]. It should be noted that such scheduling techniques do not uncover additional parallelism. Instead, they determine only how much of the detected parallelism will actually be exploited during parallel execution.

In this paper, we first present an overview of our technique, together with examples demonstrating the subpath partitioning and data flow computation. The rest of the sections then provides more details about our algorithms. We present data flow equations and a parallel algorithm for computing data flow along a linear path. We also present a technique for integrating the data flow computation of independent paths that interact by the merging and branching of paths in the control flow graph. Transformation techniques, which are applied to cyclic structures, are presented that further enhance the detection of parallelism. Experimental results comparing our technique to the parallel hybrid technique [13] that uses only the structure of the program for its parallelism are also presented.

II. Overview

Data flow analysis involves the computation of information about the flow of data along execution paths using a control flow graph. The goals of the parallel data flow analysis algorithm are to propagate the information along a path as quickly as possible and also to propagate information along different paths in parallel. We first give a general overview of a parallel algorithm that achieves the above goals. To simplify the discussion, we discuss our technique in the context of reaching definitions (i.e., a forward, union data flow problem), although our technique can be used to solve the other rapid data flow problems, including reachable uses, available expressions, and very busy expressions (i.e., forward, backward, union, and intersection).

Consider a linear chain of basic blocks. Although it appears that computing data flow for a linear path must take linear time, that is, $O(n)$ time for a path containing n basic blocks, we present a faster algorithm. From this chain, a tree can be built whose leaves represent the basic blocks, whose internal nodes represent subpaths, and whose root represents the entire data flow path. Using this tree, the computation of data flow information is carried out in two passes. In the first pass, the tree is traversed in a bottom-up fashion, and the set of definitions preserved and the set of definitions generated by each node in the tree are computed. In the second pass, starting at the root of the tree and traversing down the tree, the set of data flow items entering a node (the IN set) and the set of definitions exiting the node (the OUT set) are computed. The above process is completed in $O(\log n)$ time, because the nodes at a given level are processed in parallel (assuming that the number of processors is at least equal to the maximum number of nodes at any level and a constant time operation at each node). Fig. 1(a) shows a chain of basic blocks, and Fig. 1(b) shows the corresponding combination tree. The node labeled "1.2" represents the subpath consisting of blocks 1 and 2.

An acyclic flow graph can be viewed as consisting of several linear paths. The flow graph in Fig. 2(a) contains three paths, 1.2.3.4.5.6, 1.2.7.8.5.6, and 1.2.3.8.5.6. To compute data flow for an acyclic graph, a tree corresponding to each of these linear paths is built by combining two nodes from lower levels. Portions of paths may overlap. For example,

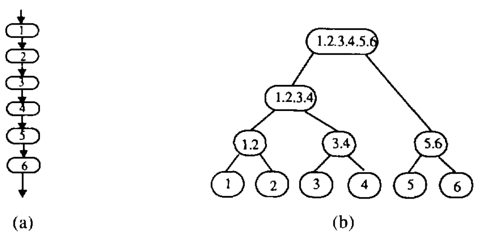


Fig. 1. Propagating data flow information along a linear path.

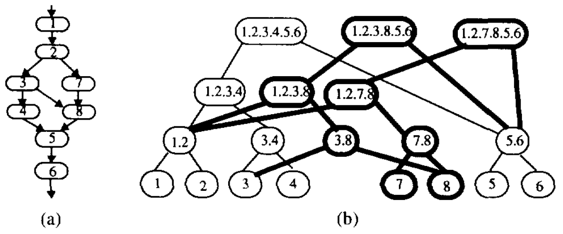


Fig. 2. Propagating data flow information in an acyclic graph.

subpath 1.2 appears in all three paths of Fig. 2(a). The nodes corresponding to the shared portions of the paths are shared among the trees constructed for different paths. This results in a combination DAG, as shown in Fig. 2(b). It should be noticed that the sharing of nodes occurs at all levels in the tree. The computation of data flow is carried out as before, with a bottom-up pass of the combination DAG followed by a top-down pass. In this process, the computation of data flow for independent paths, or independent portions of the paths, is being carried out in parallel. If a node has multiple parents, it will receive information from each of its parents. This information is combined using union and intersection operators, depending upon the data flow problem being solved. The maximum speedup that can be achieved is bounded by the length of the longest path in the program. Note that the height of the acyclic graph in Fig. 2(b) is the same as the graph in Fig. 1, because the length of the longest path is the same.

Loops introduce cycles in the flow graph. We transform the flow graph into an acyclic graph by removing the back edge of a loop and introducing additional copies of the nodes belonging to the loop (i.e., loop unwinding). The copies are introduced to ensure that the paths in the cyclic graph along which data flow information is propagated are also present in the transformed acyclic graph. Fig. 3 demonstrates the transformation of a single loop into an acyclic graph. The loop unwinding transformations are described in detail in the next section. The length of the longest path in the code will not necessarily double as a result of replication of the loop, because the computation of data flow for the replicated nodes can be carried out in parallel with the nodes following the loop. Once an acyclic graph is obtained, the combination DAG is constructed.

During the transformation of a cyclic graph into an acyclic graph, certain nodes are replicated. The data flow set of a node in the original control flow graph is the union of the data flow sets of replicated copies in the transformed graph.

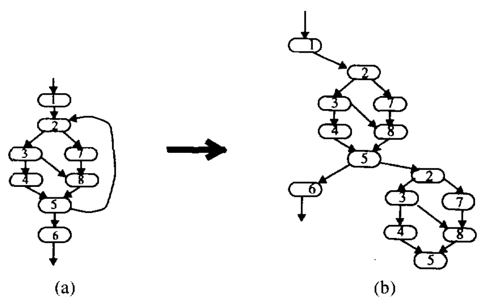


Fig. 3. Transforming a cyclic graph into an acyclic graph.

```
Given: A reducible control flow graph G = (V,E), where V is the set of basic blocks and E is the set of
edges in the flow graph
Output: In and Out for each B ε V.

Algorithm:
For each procedure in G Loop
    Eliminate Loops: The loops in G are unwound and the replicated copies of the loop
        bodies are connected to the flow graph. This results in an acyclic graph.
    Create Combination DAG: Considering all data flow paths in G create a DAG showing
        the evaluation and combination order.
    Compute Data flow: Perform computations at each node in the DAG, first Spontaneous
        and Preserved in bottom-up order, then IN and OUT in top-down order.
End For
```

Fig. 4. The parallel data flow analysis algorithm.

The major steps of the parallel data flow analysis algorithm are summarized in the algorithm shown in Fig. 4. First, the control flow graph is transformed into an acyclic graph through the unwinding of loops. Next the combination DAG for the acyclic graph is built, which is then used to compute the data flow information in parallel. Once a combination DAG has been built, it can be used repeatedly to solve the same or different data flow problems. The overall algorithm for computing data flow in parallel is summarized in Fig. 4.

III. Loop Unwinding Transformations

In this section, we discuss the transformation of a flow graph containing loops into an acyclic graph. The acyclic graph should provide the same data flow information as the original flow graph. In order to guarantee correct data flow information, we rely on the following properties.

- *Condition-1*: All acyclic paths in the original flow graph must also be present in the transformed flow graph.
- *Condition-2*: The transformed flow graph should not contain any path that was not present in the original flow graph.

The above observations were used by Kennedy [11] in developing node listings for carrying out data flow analysis of structured programs. Next we describe the transformations for handling a single loop as well as nested loops. We also prove that our transformations are in accordance with the properties described above. In the discussion of these transformations, we assume that the goal of any forward data flow analysis algorithm is to compute the data flow information immediately following each node (i.e., the OUT set for the node). The OUT set of a node is the function of the OUT sets of the node's predecessors.

Consider the flow graph in Fig. 5(a), which contains a single *repeat* loop. In the flow graph representation, the node labeled

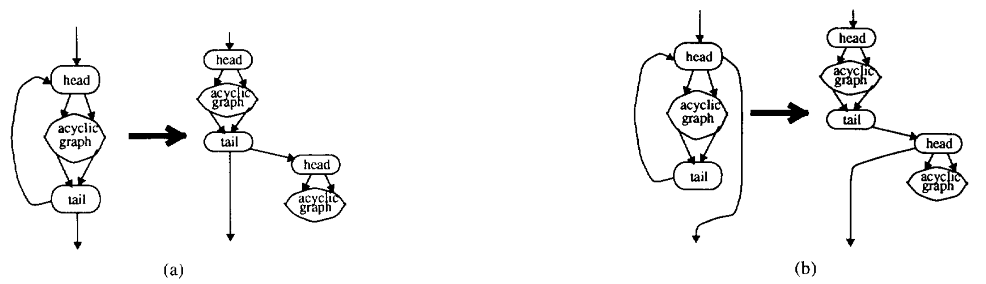


Fig. 5. Unwinding transformations for non-nested loop.

"head" represents the entry to the loop, and "tail" represents a dummy node that marks the exit of the loop. The data flow information can be propagated to all the nodes by iterating through the loop twice for a rapid data flow problem, because one pass around the loop summarizes the loop's data flow contributions. Hence, duplicating the loop once, as shown in Fig. 5(a), is sufficient to ensure the propagation of data flow information from any node in the loop to every node in the loop. As we can see, this process creates all acyclic paths present in the original control flow graph. The entire loop, except the tail of the loop, is duplicated; the tail is not duplicated, because the information arriving at the exit of the tail does not change as a result of propagating data flow information along the loop back edge. A *while* loop is handled similarly, as shown in Fig. 5(b).

Claim 1: The transformation for unwinding a single loop will lead to the computation of correct data flow information.

Proof: In order to prove this claim, we must show that the unwinding transformations in Fig. 5 satisfy the two conditions stated at the beginning of this section. Consider a pair of nodes n_i and n_j belonging to the loop. There is an acyclic path from n_i to n_j in the original control flow graph. This acyclic path is also present in the transformed graph. If this path does not go through the back edge, then the path is present in the first copy of the loop in the transformed graph. If the path goes through the back edge, then it is present from the first copy of n_i and the second copy of n_j. Thus, the acyclic path is present in the transformed graph; hence, *condition*-1 is satisfied. In the transformed graph, there is no direct edge between a pair of nodes that are not connected in the original control flow graph. Thus, no new paths are added by the transformation, and *condition*-2 is satisfied. □

The transformation of nested loops is carried out as follows. Consider the nested loops shown in Fig. 6(a), where the head and tail of each loop are directly connected by an edge. First, we handle the outermost loop by replicating the loop body and connecting the tail of one copy with the head of the newly created copy. This process creates the acyclic paths solely due to the outermost loop. Next we must create the acyclic paths for the inner loop. This process does not require replication of the inner loop, because the replication has already been carried out. The acyclic paths solely due to the inner loop are created by connecting the tail of the first copy to the head of the second copy. The resulting acyclic graph is shown in Fig. 6(b).

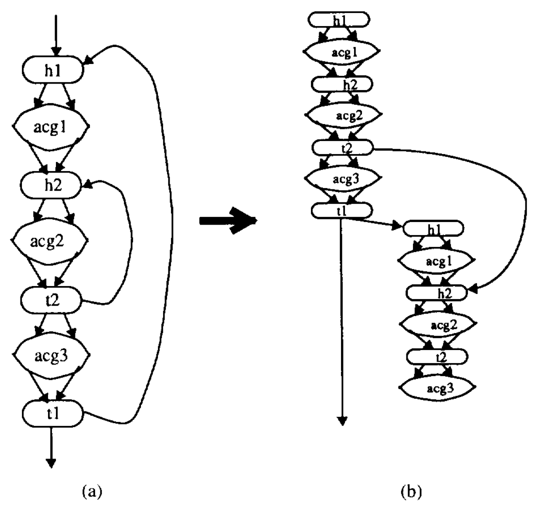


Fig. 6. Unwinding C shell loops: Head and tail connected directly.

Claim 2: The transformation for unwinding a multiply nested loop with head and tail directly connected will lead to the computation of correct data flow information.

Proof In order to show that *condition-1* is satisfied, we proceed as follows. Let us first consider the acyclic paths in the outer loop that do not traverse the back edge corresponding to the inner loop. The edge connecting t_1 to h_1 in the transformed acyclic graph creates all such paths. This directly follows from claim 1. Similarly, it also follows from claim 1 that all acyclic paths that traverse the back edge of the inner loop are created by connecting t_2 to h_2 in the transformed graph. We can also see that *condition*-2 is satisfied by this transformation because no two nodes are directly connected in the transformed graph, unless they are not directly connected by an edge in the original control flow graph. Thus, the transformation leads to the computation of correct data flow information. □

In the previous case, the exit of the loop is directly connected by the back edge to the loop entry. In some situations, the loop may contain a sequence of back edges that connects the exits of loops to the entries of loops (see Fig. 7(a)). In this situation Fig. 7(b) gives the final acyclic graph. In order to create the acyclic paths in the loop (h_1, acg_1, t_1), we replicate (h_1, acg_1); to create the acyclic paths in the loop (t_1, acg_2, t_2), we replicate (t_1, acg_2).

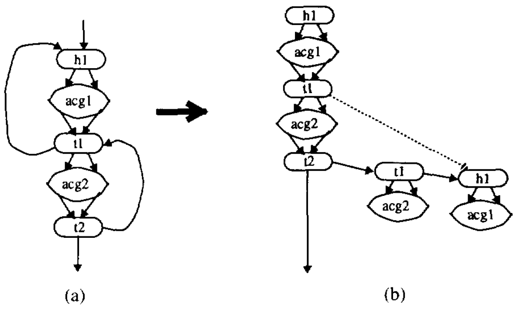


Fig. 7. Unwinding climbing loops: Head and tail connected indirectly.

Claim 3: The transformation for unwinding a multiply nested loop with head and tail indirectly connected will lead to the computation of correct data flow information.

Proof: In order to show that *condition*-1 is satisfied, we consider the following cases.

1) *Acyclic Paths Among the Nodes Without Going Through Any Back Edge:* These paths are present in the transformed graph since the entire loop body is included in the transformed graph.
2) *Acyclic Paths From* acg_2 *to* acg_2 *via Back Edge* $t_2 \rightarrow t_1$*:* These paths are included by making a copy of t_1 and acg_2 and connecting t_2 to t_1 in the transformed graph.
3) *Acyclic Paths From* acg_2 *to* acg_1 *via Both Back Edges:* These paths are also present, because there is a path starting at the first copy of acg_2 that goes through t_2 and the second copies of t_1 and h_1 leading to the second copy of acg_1 in the transformed graph.
4) *Acyclic Paths From* acg_1 *to* acg_1 *via Back Edge* $t_1 \rightarrow h_1$*:* Although loop unwinding produces multiple copies of a node, a common data set is associated with all copies. For example, the OUT set of a node in the original control flow graph is the union of the OUT sets of all of its copies in the transformed graph. Because of the unioning of the data flow sets for the replicated copies of t_1, there is an implicit path from t_1 to h_1, as indicated by the dotted edge in Fig. 7(b). Thus, the acyclic paths from acg_1 to acg_1 via back edge $t_1 \rightarrow h_1$ are represented in the transformed graph.

Thus, *condition-1* is satisfied by this transformation. For the same reasons as those given for claims 1 and 2, *condition*-2 is also satisfied. □

An overall algorithm that converts a structured cyclic flow graph into an acyclic flow graph is given in Fig. 8. This algorithm essentially selects the order in which the four transformations discussed in this section are applied. The application of unwinding transformations is preceded by the detection of loops. We identify the largest outermost loop, and, depending upon its structure, use an appropriate transformation to unwind it. The unwinding is carried out by the function *Unwind*. If an unwinding transformation can eliminate all back edges among the nodes in the current loop, then we apply the transformation. However, if this is not the case, it means that there are other loops nested within this loop that must be unwound first. This is achieved by recursively calling the function *Unwind* and removing the back edges for these loops.

```
Given: A structured control flow graph G = (V,E).
Output: An acyclic graph G′ =(V′,E′).

Algorithm:
 Procedure Unwind (L: loop)
   Let B denote the set of all backedges among the nodes in L;
   Let B′ denote the set of backedges that will be removed by applying
    one of the following transformations appropriate for L:
      T1: Transformation of Single Loop, Repeat Structure;
      T2: Transformation of Single Loop, While Structure;
      T3: Transformation of Multiple Loop, Direct edges between Head and tail; or
      T4: Transformation of Multiple Loop, Indirect edges between Head and tail.
   If B=B′ Then
      apply the appropriate unwinding transformation Ti
   Else
      B″ = B - B′
      Repeat
            find the largest nested loop L′ in L which only contains backedges from B″;
            Unwind(L′); -- this eliminates the backedges from L′
            remove the eliminated backedges from B″
      Until B″ = ∅
      remove backedges in B by applying the appropriate unwinding transformation Ti to L
      -- note that only the original nodes in L are replicated at this point.
      -- the copies created during the removal of the backedges in B″ are not replicated again.
   Endif
 End Unwind

Identify all loops in the program
While ∃ a backedge that has not been removed Do
      Select an outermost loop L = (VL,EL) in G
      Unwind (L)
Endwhile
```

Fig. 8. The loop unwinding algorithm.

```
Given: An acyclic control flow graph G = (V,E).
Output: The nodes in the combination DAG.
Algorithm:
 Procedure traverse ( node, state )
      For each child of node in the control flow graph Do
       If state = Odd Then
         If the subpath node "node.child" has not been created earlier Then
           Create the node "node.child"
           traverse ( child, Even )
         Endif
       Else /* state = Even */
         traverse ( child, Odd )
       Endif
      Endfor
 End traverse

 While paths containing multiple nodes exist Loop
    For each root node n Do  traverse ( n, odd );
    Construct the modified control flow graph - which contains the new nodes.
 Endwhile
```

Fig. 9. Construction of the combination DAG.

IV. Construction of the DAG from the Control Flow Graph

After the control flow graph has been transformed into an acyclic structure, the combination DAG is constructed. The algorithm for constructing this DAG is described in Fig. 9. In the algorithm, *Procedure Traverse* travels along all data paths in the acyclic control flow graph and generates nodes resulting from the combining of two successive nodes in the graph. An even-odd marking of nodes determines the pairs of nodes that are to be combined. The first node in a pair is marked odd, and the second is marked even. After the first application of *Traverse* to the control flow graph, all combinations containing two nodes that must be included in the combination tree are detected. The control flow graph is then transformed by replacing the original nodes with the combined nodes. Thus, the nodes in a transformed flow graph represent subpaths created by earlier node combinations. The

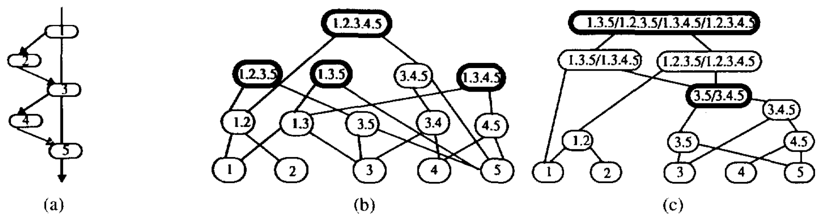


Fig. 10. Reducing the number of nodes in the combining DAG.

resulting control flow graph is then traversed again to generate the nodes at the next level of the combination DAG.[1] The process is repeated until all the nodes in the combination DAG have been generated, which occurs when the control flow graph has been reduced to single unattached nodes.

In the algorithm in Fig. 9, when merging with a path already processed, the merge point may require combining the same nodes, in which case the nodes are reused. If this is not the case, then new nodes with proper combining must be generated. However, there are at most two different combinations of shared paths. In this algorithm, two representations of a shared subpath can be created. For example, in Fig. 10(b), the two paths [1.2.3.5] and [1.3.5], which share the subpath [3.5], are derived by different combinations of nodes. Instead, we could have created a node representing the shared subpath and reused it in the derivation of other paths. We developed a modified *Traverse* procedure that enables the sharing of paths. This sharing is achieved by introducing *combining* nodes that represent multiple subpaths shared by other paths. For example, in Fig. 10(c), the combining node [3.5/3.4.5] represents two subpaths [3.5] and [3.4.5] that are shared by several paths (e.g., [1.3.5] and [1.3.4.5]). The introduction of combining nodes reduces the total number of nodes in the combination DAG. The cost is a taller DAG, bound by the number of merges found along a path. Thus, in the worst case, the height of the DAG can double if all merges occur along the same path. However, in the experiments performed, the sizes of the DAG's were significantly reduced, with the penalty being a slight change in execution time.

V. Computation of Data Flow

Given the DAG that is created from the subpaths in the control flow graph, we compute the data flow for each of the nodes. An interior DAG node can have at most two children, but multiple parents. The computed data flow at each node represents the data flowing into and out of the subpath represented by the node.

Data flow information is computed in two traversals of the DAG. The first traversal is a bottom-up traversal, and, during this traversal, a *Preserved* set, P, and a *Spontaneous* set, S, are computed for each node. The set P represents the data items that are preserved through the node; that is, if a value reaches the start of the subpath represented by the node, it reaches the end of the path. The set S at a node represents the set of data flow items that are generated within, and not killed by, the subpath represented by the node.

We assume that the set **U** is the universal set, and *Gen* and *Kill* have the usual data flow definitions for a basic block. Thus, for the leaf nodes, the S set is the *Gen* set, and the P set is the set of items that are not *Killed*. In the data flow equations, we represent a left child of a node, say, **q**, by q_l, and represent the right child of **q** by q_r. By the construction of the DAG nodes, the subpath of a left child node flows into the subpath of a right child node. The data flow equations for P and S follow:

$$P[q] = \begin{cases} U - \text{Kill}[q], \text{where } q \text{ is a leaf node,} \\ \bigcup\limits_{\text{c is a child of q}} P[c], \text{ where } q \text{ is a combining node,} \\ P[q_l] \bigcap P[q_r], \text{where } q \text{ is an interior node.} \end{cases}$$

$$S[q] = \begin{cases} Gen[q], \text{where } q \text{ is a leaf node,} \\ \bigcup\limits_{\text{c is a child of q}} S[c], \text{where } q \text{ is a combining node,} \\ (S[q_l] \bigcap P[q_r]) \bigcup S[q_r], \text{where } q \text{ is an interior node.} \end{cases}$$

For example, for a value to be in the P set of the node [1.2] in Fig. 1, representing the subpath 1.2, it must be preserved through node 1 and node 2, and thus must be in the intersection of the P sets of the two nodes. In order for an item to be in the S set of node [1.2], it must be generated in the left child of the node and preserved through the right child, or it must be generated in the right child. Thus, at the end of the first pass, the items for each node that are preserved and generated are computed.

In the second pass, which is a top-down pass, we compute the IN and OUT sets for each node, finally culminating in the computation of the IN and OUT sets for each leaf node (a basic block). IN and OUT sets of child nodes are computed from their parents. Given a node q, IN[q] is also contained in IN[q_l], OUT[q] is contained in OUT[q_r], and OUT[q_l] is contained in IN[q_r]. The OUT[q_l] is computed from its IN, S, and P sets. In computing the IN of the right child, we have to consider information flowing from all left siblings, where a left sibling is any left child of parents of the right child (see Fig. 11). The computation of the IN and OUT sets follow. Initially, for each root node in the combination DAG, IN[root] = null and OUT[root] = S[root]. In the equations, n_s represents a left sibling of n. In the following equations, we assume that n is not a child of a combining node.

$$\text{IN}[n] := \bigcup_{p_l = n} \text{IN}[p] + \bigcup_{n_s = l} \text{OUT}[l]$$

[1] In the algorithm in Fig. 9, we assume that the transformed control flow graph is constructed by the main program. However, the algorithm can be changed to build the modified control flow graph as the graph is being traversed.

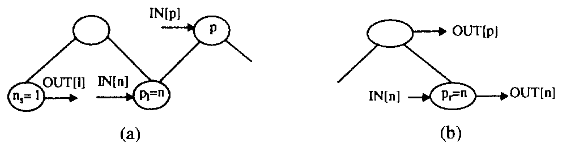


Fig. 11. The top-down pass for computing data flow.

```
Given: The Combining DAG for a control flow graph.
Output:  The Preserved and Generated Set for each node in the DAG.
Algorithm:
     Begin
          Parallel Do for nodes n at LeafLevel
               P[n] := U–Kill[q] ;
               S[n] := Gen[q] ;
          EndDo
          For i := LeafLevel-1 down to RootLevel
               Parallel do for all nodes n at level i
                    If n is a combining node Then
                         P[n] := P[c]; where c is a child of n
                         S[n] := S[c]; where c is a child of n
                    Else
                         P[n] := $P[n_l] \cap P[n_r]$ ;
                         S[n] := $(S[n_l] \cap P[n_r]) \cup S[n_r]$
                    Endif
               EndDo
          EndFor
     End
```

Fig. 12. Computing preserved and spontaneous sets in parallel.

```
Given: The P and S sets for each node in the combining DAG.
Output: The IN and OUT sets for each basic block.
Algorithm:
     Begin
          For all root nodes in the combination DAG
               In(root) := null;
               Out(root) := S(root);
          EndFor;
          For i := RootLevel+1 to LeafLevel-1
               Parallel Do for each node n at level i
                    If n is a child of a combining node p Then
                     IN[n] := IN[p]
                     OUT[n] := $(IN[n] \cap P[n]) \cup S[n]$
                    Elseif n is the left child of a node at level i-1
                     IN[n] := $\cup$ IN[p]; where n is the left child of p.
                     OUT[n] := $(IN[n] \cap P[n]) \cup S[n]$;
                    EndIf
               EndDo
               Parallel Do for each node n at level i
                    If n is the right child of a non-combining node at level i-1 Then
                     OUT[n] := OUT[n] + $\cup$ OUT[p]; where n is the right child of p.
                     IN[n] := IN[n] + $\cup$ OUT[l]; where l is a left sibling of n.
                    EndIf
               EndDo
          EndFor
     End
```

Fig. 13. Computing IN and OUT sets in parallel.

$$\mathrm{OUT}[n] := \bigcup_{p_r=n} \mathrm{OUT}[p] \text{ or}$$

$$\mathrm{OUT}[n] := (\mathrm{IN}[n] \bigcap P[n]) \bigcup S[n]$$

In the modified version of the algorithm that constructs combining nodes, the IN and OUT sets of the children of the combining nodes are computed by using the following equations.

$$\mathrm{IN}[n] := \mathrm{IN}[p], \text{ where the combining node } p \text{ is the parent of } n$$

$$\mathrm{OUT}[n] := (\mathrm{IN}[n] \bigcap P[n]) \bigcup S[n]$$

These data flow equations can be computed in parallel for each level of the DAG. The algorithms for computing S, P, IN, and OUT for reaching definitions are given in Figs. 12 and 13.

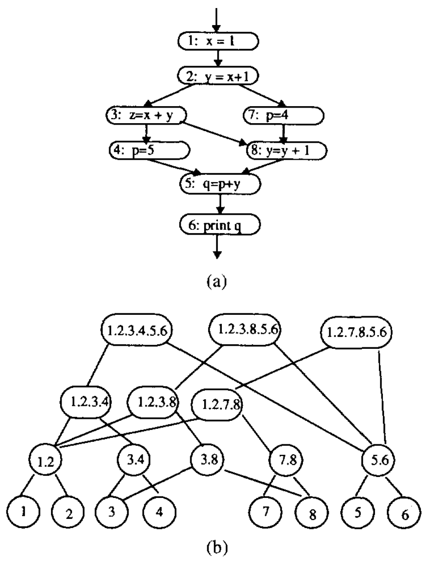


Fig. 14. Computing reaching definitions in parallel.

TABLE I
COMPUTING SETS S AND P

Node	KILL	P	S
1	{1}	{2,3,4,5,7,8}	{1}
2	{2,8}	{1,3,4,5,7}	{2}
3	{3}	{1,2,4,5,7,8}	{3}
4	{4,7}	{1,2,3,5,8}	{4}
5	{5}	{1,2,3,4,7,8}	{5}
6	{}	{1,2,3,4,5,7,8}	{}
7	{4,7}	{1,2,3,5,8}	{7}
8	{2,8}	{1,3,4,5,7}	{8}
1.2		{3,4,5,7}	{1,2}
3.4		{1,2,5,8}	{3,4}
3.8		{1,4,5,7}	{3,8}
7.8		{1,3,5}	{7,8}
5.6		{1,2,3,4,7,8}	{5}
1.2.3.4		{5}	{1,2,3,4}
1.2.3.8		{4,5,7}	{1,3,8}
1.2.7.8		{3,5}	{1,7,8}
1.2.3.4.5.6		{}	{1,2,3,4,5}
1.2.3.8.5.6		{4,7}	{1,3,8,5}
1.2.7.8.5.6		{3}	{1,7,8,5}

Next we illustrate the algorithm given in Figs. 12 and 13 by computing reaching definitions for the control flow graph shown in Fig. 14(a). The combining DAG for the control flow graph is shown in Fig. 14(b). In the first phase, we carry out the computation of sets P and S for the nodes in the DAG starting at the leaves and finishing at the root nodes. The results of this phase are given in Table I. In the next phase, we compute the IN and OUT sets for the nodes, starting at the root nodes and ending at the leaves. The results of the second phase are shown in Table II.

In the above discussion, we consider a forward union data flow problem. The other rapid data flow problems can also be computed by using the combining DAG approach. For a forward intersection problem, the union operator in the IN and OUT equations just needs to be changed to an intersection operator. For a backward flow problem, the values will flow from right to left, rather than from left to right, in combining values.

TABLE II
Computing Sets IN and OUT

Node	IN	OUT
1.2.3.4.5.6	{}	{1,2,3,4,5}
1.2.3.8.5.6	{}	{1,3,8,5}
1.2.7.8.5.6	{}	{1,7,8,5}
1.2.3.4	{}	{1,2,3,4}
1.2.3.8	{}	{1,3,8}
1.2.7.8	{}	{1,7,8}
1.2	{}	{1,2}
3.4	{1,2}	{1,2,3,4}
3.8	{1,2}	{1,3,8}
7.8	{1,2}	{1,7,8}
5.6	{1,2,3,4,7,8}	{1,2,3,4,5,7,8}
1	{}	{1}
2	{1}	{1,2}
3	{1,2}	{1,2,3}
4	{1,2,3}	{1,2,3,4}
5	{1,2,3,4,7,8}	{1,2,3,4,5,7,8}
6	{1,2,3,4,5,7,8}	{1,2,3,4,5,7,8}
7	{1,2}	{1,2,7}
8	{1,2,7}	{1,7,8}

VI. Complexity Analysis

Here we briefly describe the worst case space and run-time complexity of the combining DAG method. The space complexity of the algorithm is bounded by the size of the combining DAG. If there are n nodes in the control flow graph, there will be n leaves in the DAG. The maximum number of nodes at each level in the DAG is limited to $O(n)$. The height of the DAG is limited to $O(n)$, because the longest path in the control flow graph can have at most n nodes. The number of combining nodes cannot exceed $O(n)$, because it is limited by the number of merges in the control flow graph along a path. Thus, the worst case space complexity of the combining DAG is $O(n^2)$.

The time for using this algorithm consists of the time required to build the combining DAG and the time required to perform the data flow computation in parallel. The time spent on constructing the combining DAG is amortized across the different data flow problems being solved, because it is to be built once and reused repeatedly. Let us consider the time required to compute the data flow information. Since the data flow computation at each level is performed in parallel, the execution time is limited by the height of the combining DAG. Thus, assuming that the amount of computation performed at each node is constant, the time complexity of the parallel algorithm is $O(n)$. The worst case sequential execution time of performing data flow analysis is $O(n^2)$, even though in practice it is found to be $O(kn)$, where k is the depth of the graph. The worst case parallel execution time using $O(n)$ processors is $O(n)$. Thus, we have achieved linear speedup in the worst case execution time using our approach.

VII. Implementation and Experimentation

In order to determine the amount of parallelism detected by our algorithm for real programs, we computed the ideal execution times of our parallel algorithm and the parallel version of the hybrid algorithm, and compared their performances with the sequential hybrid algorithm. The ideal execution times were obtained by measuring the sequential execution times of each component and factoring in the parallelism detected.

In the implementation of our combining DAG technique, an adjacency matrix is used to represent the DAG for fast access to the nodes. The matrix is initialized as the control flow graph. When unwinding the loops, the control flow graph is traversed, and the matrix is updated to represent the acyclic graph representation. To share existing DAG nodes, a hash table is built to quickly search for nodes; the children of a node are used as the hash key. This dramatically speeds up the construction process.

TABLE III
Comparison of Parallel Hybrid and Combining DAG Methods

Program	Parallel Hybrid (PH)		Combining DAG (CN)		
	#scc	df_speed up	#nodes	height	df_speed up
LU factor	13	3.3	151	18	5.4
Matrix Mult	7	2.3	97	11	2.7
FFT	9	2.3	134	15	4.13
Tangnf	28	2.2	294	70	3.45

We then used the ideal parallel execution times of the data flow to determine the maximum parallelism detected by each algorithm. In the experimental results, we report the **ideal** speedup over the sequential hybrid data flow algorithm. We have run four programs obtained from various test suites and computed the ideal speedup for the parallel hybrid algorithm (PH) and our algorithm with combining nodes (CN). Table III presents the results of CN as compared with the parallel hybrid method. In the table, we give the number of strongly connected components (#scc) and the number of nodes (#nodes) and height (height) of the combining DAG. We also give the speedup, or ratio of the parallel time to the sequential time, for both algorithms. The results of our experiments indicate that our algorithm clearly detects more parallelism than the parallel hybrid algorithm, achieving ideal speedups ranging from 2.7 to 5.4.

We also implemented a version of our algorithm that did not use combining nodes (NCN). For the programs considered, the reduction in the number of nodes, using CN rather than NCN, ranged from a factor of 1.7 to as high as 10. The increase in the height of the DAG was a factor of 2. The speedup changed very marginally.

As is the case with most parallel algorithms, a setup time is required for both the hybrid and our algorithm. For example, the hybrid method needs to detect strongly connected components, whereas our algorithm must construct the combining DAG. The setup time for the above programs was less than 20% of the parallel execution time, and this time is amortized over the number of data flow computations performed. Any subsequent data flow computations would use the same combining DAG.

As mentioned previously, the times presented represent the ideal speedup and do not reflect any architectural considerations, such as communication costs. Depending on the architecture, all of the parallelism detected may or may not be exploitable. For example, in a message-passing architecture, the communication costs are significant and would require larger task sizes. These larger task sizes can be acquired by using an existing merging and scheduling technique [16]. On the other hand, fine-grained, shared memory architectures can exploit the greater amounts of the parallelism that are detected by our algorithm. Similarly, the parallel hybrid algorithm can be implemented by using the same type of scheduling techniques. However, the smallest task sizes correspond to the strongly connected components of a control flow graph and can be increased in size only by these methods. The advantage of our approach is the flexibility provided in terms of constructing tasks of various grain sizes.

REFERENCES

[1] F. Allen and J. Cocke, "A program data flow analysis procedure," *Commun. ACM*, vol. 19, no. 3, pp. 137–147, Mar. 1977.

[2] T. R. Allen and D. A. Padua, "Debugging Fortran on a shared memory machine," *Proc. Int. Conf. Parallel Processing*, 1987, pp. 721–727.

[3] D. Callahan, K. Kennedy, and J. Subhlok, "Analysis of event synchronization in a parallel programming tool," *Proc. 2nd ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, 1990, pp. 21–30.

[4] P. Emrath and D. A. Padua, "Automatic detection of nondeterminacy in parallel programs," *Proc. ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, 1988, pp. 89–99.

[5] P. G. Frankl and E. J. Weyuker, "An applicable family of data flow testing criteria," *IEEE Trans. Software Eng.*, vol. 14, pp. 1483–1498, Oct. 1988.

[6] T. Gross, A. Zobel, and M. Zolg, "Parallel compilation for a parallel machine," *Proc. SIGPLAN'89 Conf. Programming Language Design and Implementation*, 1989, pp. 91–100.

[7] R. Gupta, L. L. Pollock, and M. L. Soffa, "Parallelizing data flow analysis," *Workshop on Parallel Compilation*, Kingston, ON, Canada, 1990.

[8] M. J. Harrold and M. L. Soffa, "Interprocedural data flow testing," *Proc. Software Testing, Analysis, and Verification Symp.*, 1989, pp. 158–167.

[9] J. B. Kam and J. D. Ullman, "Global data flow analysis and iterative algorithms," *J. ACM*, vol. 23, no. 1, pp. 158–171, Jan. 1976.

[10] J. B. Kam and J. D. Ullman, "Monotone data flow analysis frameworks," *Acta Informatica*, vol. 7, pp. 305–317, 1977.

[11] K. W. Kennedy, "Node listings applied to data flow analysis," *Conf. Record 2nd ACM Symp. Principles of Programming Languages*, 1975, pp. 10–21.

[12] Y.-F. Lee, T. J. Marlowe, and B. G. Ryder, "Experiences with a parallel algorithm for data flow analysis," *J. Supercomputing*, vol. 5, pp. 163–188, Oct. 1991.

[13] T. J. Marlowe and B. G. Ryder, "An efficient hybrid algorithm for incremental data flow analysis," *Conf. Record ACM Symp. Principles of Programming Languages*, 1990, pp. 184–196.

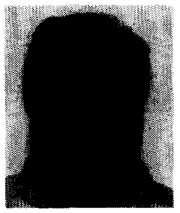
[14] B. Miller and J. D. Choi, "A mechanism for efficient debugging of parallel programs," *Proc. SIGPLAN'88 Conf. Programming Language Design and Implementation*, 1988, pp. 135–144.

[15] B. Ryder, "ISMM: Incremental software maintenance manager," *Proc. Conf. Software Maintenance*, 1989, pp. 142–164.

[16] V. Sarkar, "Compile time partitioning and scheduling of parallel programs," *Proc. SIGPLAN Symp. Compiler Construction*, 1986, pp. 17–26.

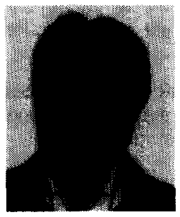
[17] V. Seshadri, D. B. Wortman, M. D. Junkin, S. Weber, C. P. Yu, and I. Samll, "Semantic analysis in a concurrent compiler," *Proc. SIGPLAN'88 Conf. Programming Language Design and Implementation*, 1988, pp. 298–312.

[18] A. Zobel, "Parallel compiler optimization," *Workshop on Parallel Compilation*, Kingston, ON, Canada, 1990.



R. W. Kramer received the B.S. degree in mathematics and computer science from Youngstown State University, Youngstown, OH, USA, in 1987, and the M.S. degree in computer science from the University of Pittsburgh, PA, USA, in 1991.

He is a doctoral student in the Department of Computer Science, University of Pittsburgh. His research focuses on parallel and incremental techniques for performing rapid genetic linkage analysis.



R. Gupta received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, New Delhi, India, in 1982, and the Ph.D. degree in computer science from the University of Pittsburgh, PA, USA, in 1987.

Currently, he is an Associate Professor in the Department of Computer Science, University of Pittsburgh. He was a Senior Member of Research Staff in the Computer Architecture and Programming Systems Group at Philips Laboratories from 1987 to 1990. His primary areas of research interest include parallelizing compilers, parallel architectures, implementation of programming languages, and software tools.

Dr. Gupta received the National Science Foundation Presidential Young Investigator Award in 1991. He serves as an Associate Editor of the *Journal of Parallel Computing*, and is a program committee member of the ACM SIGPLAN'94 Programming Language Design and Implementation Conference. He is a member of the Association for Computing Machinery, SIGPLAN, SIGARCH, and the IEEE Computer Society.

M. L. Soffa received the Ph.D. degree in computer science from the University of Pittsburgh, PA, USA, in 1977.

Since 1977, she has been a faculty member at the University of Pittsburgh, and is currently a Professor in the Department of Computer Science there. Since 1991, she has also served as the Dean of Graduate Studies in Arts and Sciences. Her research interests include language implementation, parallelizing compilers, program analysis, and software tools.

Dr. Soffa currently serves on the editorial boards of *ACM Transactions on Programming Languages and Systems*, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, and the *International Journal of Parallel Programming and Computer Languages*.