Region Scheduling: An Approach for Detecting and Redistributing Parallelism

RAJIV GUPTA AND MARY LOU SOFFA

Abstract—**In developing compiler techniques for programs targeted for parallel execution, it is imperative that a program representation be utilized that not only facilitates the detection and scheduling of parallelism but also easily enables program transformations that increase opportunities for parallelism. These requirements are the driving force behind region scheduling, a technique applicable to both fine grain and coarse grain parallelism. This technique employs a program representation that divides a program into regions consisting of source and intermediate level statements and enables the expression of both data and control dependencies. Guided by estimates of the parallelism present in regions, the region scheduler redistributes code, thus providing opportunities for parallelism in those regions containing insufficient parallelism compared to the capabilities of the executing architecture. The program representation and the transformations are applicable to both structured and unstructured programs, making region scheduling useful for a wide range of applications. The results of experiments conducted using the technique in the generation of code for a reconfigurable long instruction word architecture are presented. The advantages of region scheduling over trace scheduling, another technique for transforming and detecting fine grain parallelism in programs, are discussed.**

Index Terms—**Code optimizations, code scheduling, parallelism detection, program dependence graph, program transformations, trace scheduling.**

I. Introduction

An important compiler component for parallel architectures is a technique that detects and schedules the parallelism in a sequential program, possibly by applying code transformations to effectively utilize the system resources. This process of detecting and scheduling parallelism is done by examining the code for fine grain operations (i.e., parallel operations within and among source statements) and/or coarse grain operations (e.g., vector operations or loop parallelization), depending on the target architecture. Coarse grain parallelism is best detected using the program source code while the detection of fine grain parallelism usually requires an intermediate level program representation.

One problem in the development of scheduling techniques for both levels of granularity is that of finding sufficient parallelism to utilize all of the system resources. Programs employing coarse grain parallel operations such as vectors may have a number of scalar operations that are done serially, thus reducing the overall benefits of vectorization. Basic blocks, which are straight-line code segments with a single entry and a single exit, are typically used in detecting fine grain parallelism and may be too small to contain sufficient parallelism for the available processors.

To increase the size of the code considered for fine grain parallelism, straight line code segments can be combined into one segment. By doing this, statements possibly requiring different control conditions may be included in the combined segment. This complicates the scheduling process, for not only is more global data flow information needed but so is information about the contained control dependencies. Another approach to the above problem is to intermix the execution of coarse grain parallel operations with the fine grain operations, assuming architectural support of each type. To do this, some unusable coarse grain parallelism can be converted to fine grain parallelism. The execution of operations resulting from this conversion would then be intermixed with the execution of sequential operations to achieve a faster overall schedule.

In this paper, we present a technique called region scheduling that employs both of the above approaches in attempting to effectively schedule the parallelism in a program. An intermediate program representation is employed that enables the detection of both coarse grain and fine grain parallelism in programs. The representation is an extended form of the Program Dependence Graph (PDG) [5], which divides a program into regions containing statements requiring the same control conditions. Thus each region consists of one or more straight line code segments. Guided by the estimates of the parallelism present in the program regions, the region scheduler repeatedly transforms the extended PDG, uncovering potential parallelism in the process until an estimate of the parallelism in each region matches the parallel capabilities of the underlying architecture, or no transformations are applicable. The transformations defined for region scheduling can redistribute fine grain parallelism among regions through the transfer of code from one region to another and convert coarse grain parallelism to fine grain parallelism. Thus, excess parallelism from one region can be transferred to another region with insufficient parallelism. The

Manuscript received August 1, 1988; revised August 8, 1989. Recommended by M. Evangelist. This work was supported in part by the National Science Foundation under Grant CCR-8801104 to the University of Pittsburgh.

R. Gupta is with Philips Laboratories, 345 Scarborough Road, Briarcliff Manor, NY 10510.

M. L. Soffa is with the Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260.

IEEE Log Number 8933742.

0098-5589/90/0400-0421$01.00 © 1990 IEEE

representation and the transformations are applicable to both structured and unstructured programs, thus making region scheduling useful for a wide range of applications. The technique is architecture independent and only requires parallelism at a fine unit of granularity.

A compiler that employs region scheduling consists of three main phases. During the first phase, the extended PDG is constructed, which is then used to perform traditional global optimizations as well as detect vectorizable and loop invariant computations. Transformations such as loop interchange, expansion of a scalar to a vector variable and node splitting that make code vectorizable can be applied [2], [11]. In the second phase, the region scheduler performs transformations on the extended PDG to increase parallelism. In the final phase, transformations such as height reduction of data flow dependencies can be applied, followed by the identification of parallelism and the generation of machine instructions. All three phases use only the extended PDG as the program representation.

In subsequent sections, we first present an overview of the PDG representation and then describe the extensions for region scheduling. Next the transformations and the algorithm used to apply the transformations are detailed. Results of some experiments conducted to evaluate the performance of region scheduling are presented. The merits of region scheduling and a comparison with trace scheduling [6], a related and currently used technique, are discussed.

II. Background

Techniques for the detection of both coarse and fine grain parallel operations have been developed to take advantage of various parallel architectures. These techniques include the detection of coarse grain parallelism useful in generation of code for loosely coupled multiprocessor systems. Coarse grain parallelism found in sequential programs is mainly in the form of vectorizable computations. Considerable research attention has been devoted to the detection of vectorizable loops in Fortran programs. As an example, Parafrase, a compiler developed at Illinois for vector machines and multiprocessor systems, relies on global data dependence information and transformations of the source code to produce highly parallel code [13]. In work done by Allen and Kennedy [2], automatic techniques to carry out loop interchanges resulting in vectorizable code are presented. The PTRAN project at IBM is also aimed at exploiting coarse grained parallelism in Fortran programs [3], [15]. In each of the above major research efforts, only coarse grain parallelism was considered, as the target architectures were not designed for fine grain parallelism.

Research in the detection and utilization of fine grain parallelism has also received some attention. In the detection of fine grain parallelism, Sites [16] developed code reordering algorithms for straight line code to obtain improved performance for the scalar pipelined unit of the Cray. Techniques to reorder code in a basic block to obtain improved performance for MIPS, a pipelined reduced-instruction-set processor with multiple functional units, were developed by Hennessy and Gross [10]. The major drawback of such work is that the reordering of code is done on a per basic block basis. The basic blocks in programs are usually small and hence little parallelism can be found, with the result that not much improvement in speed can be obtained.

A technique that has effectively tackled the problem of detecting fine grain parallelism across basic blocks is trace scheduling which uses a control flow graph representation of a program [1], [4], [6]. The trace scheduler, in an attempt to increase the number of statements and thus the opportunity for parallel operations, uses the control flow graph to trace a path consisting of a number of basic blocks from which to schedule code. It repeatedly traces out paths and passes each path to a code generator for machine code translation. The operations in the trace are reordered to generate an efficient schedule of parallel instructions. In this scheme, the choice of traces is crucial to achieving good performance. The traces are chosen based on compile-time determination of execution frequency estimates of the statements. The underlying assumption of trace scheduling is that the most likely execution paths through a program can be predicted at compile time, and for this reason, it is oriented towards scientific programs. Trace scheduling is not expected to be successful for programs whose control structures are not simple and predictable. Another drawback of trace scheduling is that it does not take into account the processing capability of the system while constructing traces.

Although the control flow graph is the traditional program representation used by compilers, a newly developed representation is the Program Dependence Graph (PDG), developed by Ferrante, Ottenstein, and Warren [5]. This program representation expresses both control and data dependencies and can be utilized for efficiently performing traditional compiler optimizing transformations and vectorization transformations. The PDG also permits the incremental data flow update after each transformation. The PDG is summarized in the next section, as this representation forms the basis for the representation used in region scheduling.

III. The Program Representation

The program representation used in region scheduling is an extension of the PDG [5]. We first present a brief overview of the PDG and then discuss the extensions.

A. *Program Dependence Graph*

The PDG is a graph representation of a program that expresses both relevant control and data dependencies in a program. The nodes in the graph are statements and predicate expressions and the edges represent the dependencies. A statement S_i is control dependent upon a predicate P_j if the value of P_j immediately controls the execution of S_i. The control dependencies are derived from the control flow graph through control flow analysis. A data dependence exists between two statements if a vari-

able used in one of the statements will have an incorrect value if the order in which the two statements are executed is reversed [11]. Data flow analysis on the control flow graph is used to compute all the data dependencies in a program. The PDG representation allows uniform treatment of data and control dependences which makes transformations such as vectorization easy to perform. At the same time, the hierarchical nature of the representation allows treatment of control and data dependences at separate levels.

A control dependence subgraph is constructed as part of the PDG and is useful when performing transformations that alter the control flow structure of a program. In order to determine whether a change in the control structure can be made or not, data dependency information is needed which is also available through the PDG. Reordering of statement and predicate nodes can only be carried out if no data dependencies are violated. The PDG supports incremental update of data dependency information and thus, supports the incremental application of optimizations.

In the control dependence subgraph of the PDG, the statements and predicate expressions are represented as nodes and the control conditions for their execution as the edges between the nodes. There are three kinds of nodes in this graph: statement nodes (S_i), predicate nodes (P_i), and region nodes (R_i). A region node points to a set of nodes representing parts of a program that require the same set of control conditions for their execution, and the edges connecting different regions show the flow of control. Depending on the level of granularity desired, a statement node can represent either an intermediate level statement or a high-level nonconditional statement. Fig. 1 shows the control dependence subgraph for the following sequence of statements. Region node R_1 points to region R_8, which represents loop L_1, and predicate P_1 indicating that L_1 and P_1 require the same set of control conditions for their execution.

L_1: for $i = 1$ to N do X_i;
if P_1 then S_1;
  if P_2 then S_3; go to $<$ label $>$
   else if P_3 then $<$ label $>$: S_4
     else S_5
    endif
  endif
 else S_2
endif

B. Extensions to the Program Dependence Graph (EPDG)

To utilize the PDG for applying transformations to redistribute statements and increase parallelism in some regions, a number of extensions are made to the PDG. In particular the control dependence subgraph (CDG) of the PDG is extended (EPDG) for use in region scheduling as follows:

1) The nodes pointed to by a region or predicate node are ordered. When nodes are to be moved from region R_i to R_j, the nodes in R_i that are closest to R_j are considered first. The ordering of the nodes in each region enables examination of the nodes in the desired order. The order also helps in determining the data dependency information needed to ascertain whether a statement node can be moved from one point in the CDG to another. The order of the nodes corresponds to the order of the statements in the source program. The leftmost node corresponds to the part of the program that occurs first and the rightmost node to the part of the program that occurs last for the region.

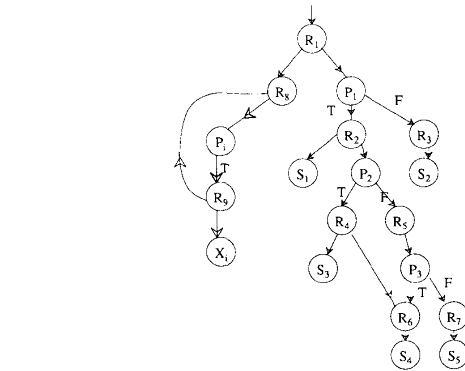


Fig. 1. Control dependence graph.

2) To move a subgraph defined by a region node to another region, we must ascertain that the two subgraphs do not intersect, and thus we define a "structure property" for the nodes. Each node in the graph is marked as *structured* if the set of statements represented by the subgraph rooted at the node is structured. To do this, loop-back edges must be distinguished from the rest of the edges in the CDG. A loop-back edge is essentially an edge in the CDG that represents the flow of control between successive iterations of a loop [1]. The edge from region R_9 to R_8 in Fig. 1 is a loop-back edge. A node is structured if after removing all loop-back edges in the graph the following conditions hold for each of its child nodes: a) the child node is structured; and b) the child node has exactly one edge pointing to it. For example, in Fig. 1 node R_6 is structured but node R_4 is unstructured. The structured property of the nodes is essential to determine the applicability of some code motion transformations described in Section IV.

3) Loops consisting only of statement nodes are suitable for unrolling. Such loops are distinguished from the rest of the loops by labeling their representative region nodes by L_i.

4) Each region node is marked with an estimate of potential parallelism in the region. This allows the application of the transformations until the region has either enough potential parallelism to utilize the system resources fully or no more transformations can be applied. An estimate of the amount of parallelism in a region is used to decide which regions should be transformed. The parallelism δ_i present in a region R_i is defined as the ratio

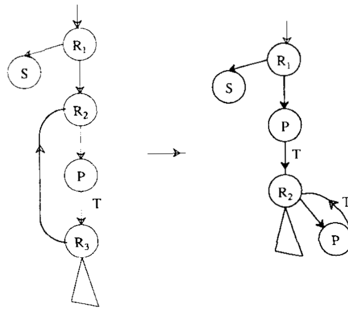


Fig. 2. Duplicating a while loop predicate.

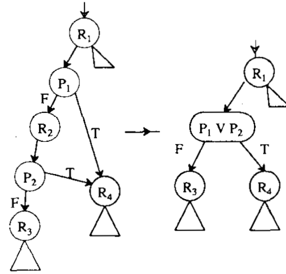


Fig. 3. Combining predicates.

O_i/D_i, where O_i is the number of operations in the region and D_i is the length of the longest data dependency chain in the region. The length of the tallest path can be found by examining the data dependences in a region. The value δ_i is an estimate that essentially indicates, on an average, the number of operations that can be performed in parallel at any given time. The maximum number of operations that the system can physically perform in parallel, as determined by the architecture, is δ_s. The goal of the scheduler is to make δ_i approach δ_s for all regions so that the system resources are utilized efficiently.

5) The aim of region scheduling is to construct regions with significant amounts of parallelism. Therefore an attempt is made to construct a CDG which has a small number of large regions. Fig. 2 shows how the evaluation of a while loop predicate may be carried out in parallel with some of the operations preceding and within a loop if the loop predicate is duplicated, essentially creating a repeat loop. The transformation shown in Fig. 2 reduces the number of regions in the CDG by one. Fig. 3 shows how predicates P_1 and P_2 may be combined to reduce the number of regions in the CDG. This allows evaluation of the combined predicates in parallel. Architectural support should be provided to enable ignoring of traps arising due to errors in P_2 in the event P_1 evaluates to false. Thus, in constructing the EPDG, while loops are represented as repeat loops and the predicates are collapsed, as shown in Fig. 2 and Fig. 3.

IV. Transformations and Updates to EPDG

The transformations performed by the region scheduler modify code that is either a single statement node, an entire subgraph rooted at a region node, or a predicate node. Data flow information is provided in the EPDG to check that the transformations are legal before being applied. Various transformations described in this paper rearrange the order in which the statement nodes appear. Reordering of statement and predicate nodes can only be carried out if no data dependencies are violated. The incremental update of the information after a transformation has been applied is also done [5], [14]. The three basic kinds of transformations applied by the region scheduler are loop transformations, region copying and collapsing transformations and forward/backward code motion transformations. The loop transformations include loop unrolling and invariant code motion. The region copying and collapsing transformations create larger regions by merging two regions, each with insufficient parallelism. The forward/backward code motion transformations move code from one region to a lower/higher region in the graph with insufficient parallelism. Application of some of these transformations to parts of the EPDG requires updating of other parts of the EPDG to maintain the semantics of the program. The updating involves the copying of code to other regions so that the code is executed under exactly the same conditions as it was before a transformation was applied. First the transformations are discussed in detail and then the updating procedures are described. The order of applying the transformations is given in Section V.

A. Loop Transformations

The two loop related transformations performed are loop unrolling and invariant code motion. Both transformations increase the parallelism in region R_i by adding code to it.

τ_{unroll} Unrolls m iterations of the loop in region L_i and puts them in region R_i, the immediate parent of L_i (i.e., $R_i, L_i^n \rightarrow R_i + L_i^m, L_i^{n-m}$) [see Fig. 4(a)]. This results in an increase of parallelism in region R_i. Loop unrolling is also used to increase the parallelism within the loop body L_i (i.e., $L_i \rightarrow (L_i + L_{i+1})^{n/2}$).

τ_{invar} Moves a loop invariant computation S_2 in region R_j to region R_i outside the loop (i.e., $R_j \rightarrow R_i$) [see Fig. 4(b)]. This transformation may require updating of the EPDG which is described later in this section.

B. Forward/Backward Code Motion

Forward and backward code motion transformation move code either up the graph or down the graph to increase the parallelism present in some region R_i. These transformations are always applicable to *structured* nodes, data dependencies permitting. To move a subgraph rooted at an unstructured node requires analysis to first determine if the unstructured node and the region node R_i have common descendants. The movement of such a node would then require additional analysis to ascertain the necessary modifications of the control dependence graph if common

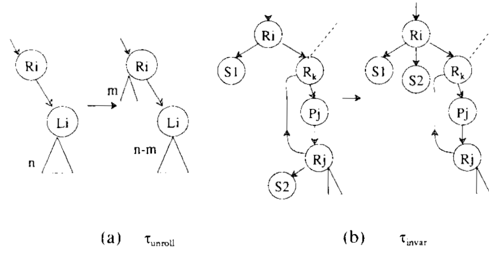


Fig. 4. Loop transformations. (a) τ_{unroll}. (b) τ_{invar}.

descendants did exist. Thus, if there are no common descendants, the unstructured node can be moved; otherwise, the node is not moved. For example, in Fig. 1, the unstructured subgraph rooted at P_3 will not be moved from region R_5 to region R_2.

τ_{move} Moves code forward or backward to region R_i from an adjacent region R_j (i.e., $R_j \rightarrow R_i$). There are two forward (τ^{f1}_{move}, τ^{f2}_{move}) [see Fig. 5(a)-(i), 5(a)-(ii)] and two backward (τ^{b1}_{move}, τ^{b2}_{move}) [see Fig. 5(b)-(i), 5(b)-(ii)] move transformations. The transformations τ^{f1}_{move} and τ^{b1}_{move} are applied when the regions R_j and R_i are connected directly by an edge. The transformations τ^{f2}_{move} and τ^{b2}_{move} are applied when the regions are connected through a predicate node P_i. These transformations are applied if they either increase the parallelism in both the regions or if they transfer excess parallelism in one region to another region with insufficient parallelism.

When the forward transformation τ^{f1}_{move} in Fig. 5(a)-(i) is applied, the code moved from R_j to R_i will be executed even when the control to R_i does not come through R_j. Similarly when transformation τ^{b2}_{move} in Fig. 5(b)-(ii) is applied, the code being moved from R_j to R_i is executed irrespective of whether the predicate P_i evaluates to true or false. However, this will not reduce the execution speed of the program as in either case the transformation is applied only when R_i has insufficient parallelism and hence some processors in the system are idle during the execution of R_i. The forward transformation τ^{f2}_{move} in Fig. 5(a)-(ii) and backward transformation τ^{b1}_{move} in Fig. 5(b)-(i) require further updating of the graph which is described at the end of this section.

C. Region Copying and Collapsing

Region copying and collapsing transformations take a region R_i with insufficient parallelism and either copy its code into each of its parent regions and thus eliminate the region or merge it with another region. These transformations eliminate the need for a branch instruction, increasing the speed of the operation if processors are pipelined.

τ_{copy} Creates a copy of all the nodes in the structured subgraph rooted at region node R_i in each of its parent nodes. This is achieved by applying the following transformations:

1) If a parent of R_i is a region node (R_j) then R_j is made the parent of copies of all the nodes in region R_i [see Fig. 6(a)].

2) If a parent of R_i is a predicate node (P_j) then a copy of the subgraph rooted at R_i is made and P_j is made the parent of this new subgraph [see Fig. 6(b)].

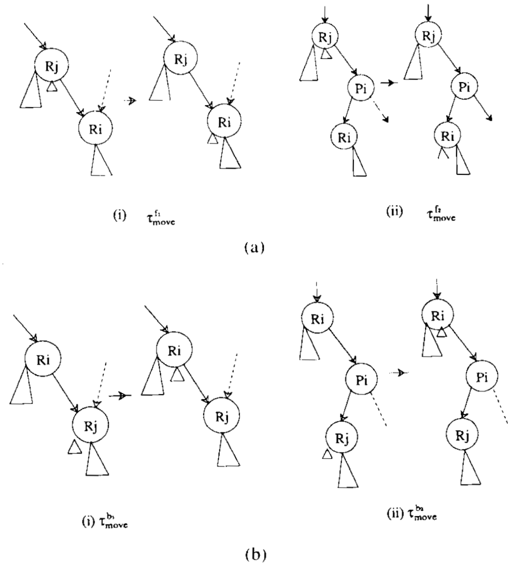


Fig. 5. Transformation τ_{move}: ($R_j \rightarrow R_i$). (a) Forward code motion. (b) Backward code motion.

These transformations make the program more structured by eliminating unconditional branches. Applying the above transformations may create redundant region nodes which are removed by another transformation.

τ^1_{merge} Merges P_k, R_j with R_i, where P_k is the only parent of R_i and R_j, to form statement node S_k (i.e., R_i, P_k, $R_j \rightarrow S_k$) (see Fig. 7). After τ^1_{merge} has been applied, the operations in P_k, R_i, and R_j are treated as a single unit for the purpose of applying further transformations. The predicate and the two regions are executed in parallel and depending upon the value of the predicate, the result from executing one of the regions is discarded. This transformation can only be used if the architecture allows discarding of values; otherwise, the conditional branch instruction cannot be deleted and this transformation is not applicable.

τ^2_{merge} Merges region R_i with region R_j forming a single region R_k (i.e., R_i, $R_j \rightarrow R_k$) (see Fig. 8). This transformation deletes redundant region nodes that may be created due to the application of other transformations.

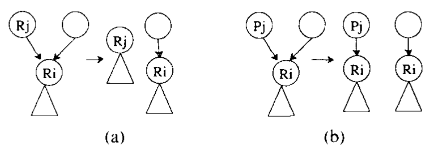


Fig. 6. Transformation τ_{copy}.

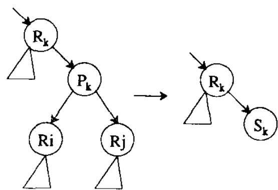


Fig. 7. Transformation τ^1_{merge}: $(R_i, P_k, R_j \rightarrow S_k)$.

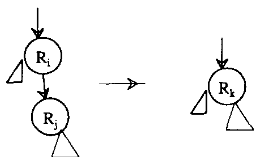


Fig. 8. Transformation τ^2_{merge}: $(R_i, R_j \rightarrow R_k)$.

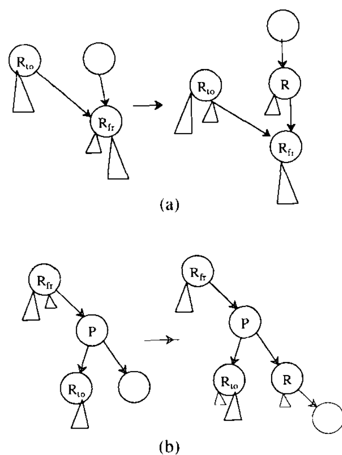


Fig. 9. Updating the EPDG.

D. Updates Due to Transformations

The application of certain transformations require updating of the EPDG to ensure preservation of program semantics. This situation occurs when relocated code would not be executed under conditions that it should be executed. When code is moved from region R_{fr} to region R_{to} using transformations τ_{invar} or τ^{b1}_{move}, copies of the code have to be created for each of the parents of R_{fr} other than R_{to}, as shown in Fig. 9(a). When transformations τ^{f2}_{move} is applied, a copy of the moved code has to be made as shown in Fig. 9(b). Applying transformation τ_{copy} and the above updates may cause redundant region nodes to be created in the graph. These are eliminated by subsequently applying transformation τ^2_{merge}.

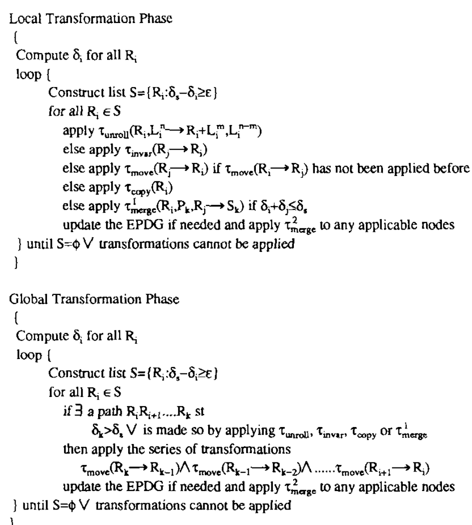
V. Algorithm for Applying the Transformations

The transformations described can be applied repeatedly to the EPDG to increase the parallelism present in regions. The repeated application is required because application of one transformation may enable application of others in subsequent steps. In order to find all of the parallelism in a program that the above transformations can uncover, the transformations can be repeatedly applied as long as they continue to increase the parallelism in the regions. However, in practice the system on which the program is to be executed will have a finite amount of resources, thus limiting the amount of parallelism that can be exploited. For this reason an algorithm which applies the transformations until regions have sufficient parallelism for the architecture under consideration is developed.

The algorithm in Fig. 10 summarizes the manner in which the transformations are applied by the region scheduler. In this algorithm, transformations are applied as long as regions with insufficient parallelism exist. Thus instead of unrolling a loop a fixed, predetermined number of times, as is done in traditional compilers, it is unrolled only if more parallelism is needed in a region. In ordering the transformations, those transformations that only consider adjacent region nodes, and thus need only a local view of the graph, are first performed. Next, transformations requiring a global view of the graph are applied, during which if a region node (R_i) with insufficient parallelism is connected along a path to a region node (R_k) with excess parallelism, transformation τ_{move} is applied repeatedly to move excess parallelism from R_k to R_i. If region R_k does not contain excess parallelism then excess parallelism is created by applying one of the transformations τ_{unroll}, τ_{invar}, τ_{copy}, or τ^1_{merge} locally. The path chosen is the smallest one for which the global transformation can be applied. In a structured program each region node has at most one parent and in an unstructured program, application of transformation τ_{copy} reduces the number of parents of a region node with insufficient parallelism to one whenever possible. For example, in Fig. 6(a) after the application of τ_{copy}, region R_i has a single parent. This limits the number of paths along which transformation τ_{move} can be applied to one in most cases.

The transformations are first performed locally and then globally because more overhead is involved in applying global transformations. The order in which the transformations are applied has been chosen based on their effectiveness and overhead. τ_{unroll} and τ_{invar} are applied before τ_{move} because they are less expensive to perform. The transformation τ_{copy} is applied next because, although effective in increasing parallelism and reducing branches, it causes more duplication of code than τ_{move}. Transformation τ^1_{merge} is applied last as it only reduces the number of branch instructions in a program.

Fig. 11 shows how the control dependence subgraph of Fig. 1[11(a)] can be modified after the transformations



```
Local Transformation Phase
{
 Compute δi for all Ri
 loop {
      Construct list S={Ri:δs−δi≥ε}
      for all Ri ∈ S
         apply τunroll(Ri,Li^n ⟶ Ri+Li^m,Li^(n−m))
         else apply τinvar(Rj ⟶ Ri)
         else apply τmove(Rj ⟶ Ri) if τmove(Ri ⟶ Rj) has not been applied before
         else apply τcopy(Ri)
         else apply τ^1merge(Ri,Pk,Rj ⟶ Sk) if δi+δj≤δs
         update the EPDG if needed and apply τ^2merge to any applicable nodes
 } until S=φ ∨ transformations cannot be applied
}

Global Transformation Phase
{
 Compute δi for all Ri
 loop {
      Construct list S={Ri:δs−δi≥ε}
      for all Ri ∈ S
         if ∃ a path RiRi+1....Rk st
               δk>δs ∨ is made so by applying τunroll, τinvar, τcopy or τ^1merge
         then apply the series of transformations
            τmove(Rk ⟶ Rk−1)∧τmove(Rk−1 ⟶ Rk−2)∧ ......τmove(Ri+1 ⟶ Ri)
         update the EPDG if needed and apply τ^2merge to any applicable nodes
 } until S=φ ∨ transformations cannot be applied
}
```

Fig. 10. Region scheduler.

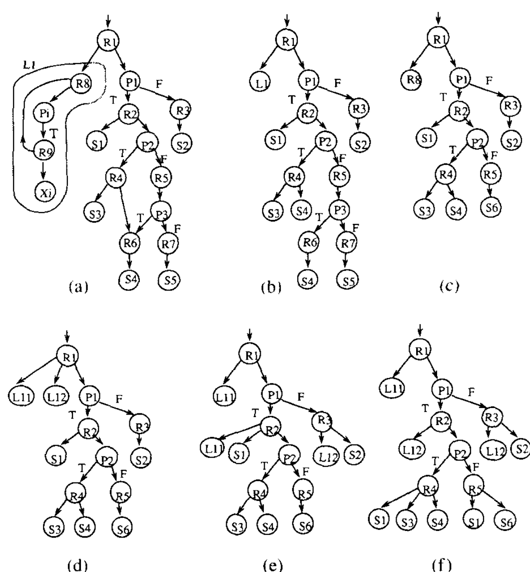


Fig. 11. Applying transformations.

have been applied. Assuming that R_6 has insufficient parallelism, transformation $\tau_{copy}(R_6)$ is first applied and then $\tau^1_{merge}(R_6, P_3, R_7)$ creating node S_6 during the local transformation phase [see Fig. 11(b) and (c)]. Then assuming R_4 has insufficient parallelism during the global transformation phase, the series of transformations $\tau_{unroll}(R_1, L_1^n \rightarrow R_1 + L_1^m, L_1^{n-m})$, $\tau_{move}(R_1 \rightarrow R_2)$ and $\tau_{move}(R_2 \rightarrow R_4)$ are applied [see Fig. 11(d), (e), and (f)]. In this series of transformations, the excess parallelism in R_1 created by unrolling the loop L_1 is transformed to R_4 through R_2. It should be noted that the portion of code moved by transformations $\tau_{move}(R_1 \rightarrow R_2)$ and $\tau_{move}(R_2 \rightarrow R_4)$ is not the same. S_1 can only be moved to R_4 after excess parallelism is created in R_2 through loop unrolling and code motion from R_1 to R_2. The resulting EPDG has five regions as opposed to seven in the untransformed graph. Moreover the regions in the transformed graph are much larger than the ones in the original graph and thus are likely to contain more parallelism. The code resulting from the transformations is as follows:

L_{11}: X_1; X_2; $\cdots$ $X_{N/2}$;
if P_1 then L_{12}: $X_{N/2+1}$; $\cdots$ X_N;
    if P_2 then S_1; S_3; S_4;
        else S_1; S_6;
    endif
  else L_{12}: $X_{N/2+1}$; $\cdots$ X_N;
    S_2;
endif

After the control dependence subgraph has been transformed and parallelism redistributed, data dependence subgraphs for the individual regions are constructed, by performing data flow analysis. The data dependence subgraphs are linked by the data flow information. The code generator can now schedule the code in the individual regions. Traditional optimizations such as common subexpression elimination, constant folding, copy propagation, and induction variable simplification and forward substitution may be performed on the regions. Tree height reduction [12] techniques can be applied to increase the parallelism in the data dependence subgraphs before code generation.

VI. Experimental Results

The local transformation phase of region scheduling was implemented on a VAX 11/780 as part of a prototype compiler for a reconfigurable long instruction word (RLIW) architecture [8], [9], which supports fine grain parallelism. The long instruction word enables simultaneous initiation of a set number of fine grain operations. The prototype compiler was used to perform experiments in order to gain some insight into the effectiveness of region scheduling. In these experiments, δ_s was set to eight to represent eight simultaneous operations. First, the improvement in performance due to detection of parallelism using individual regions instead of individual basic blocks was evaluated. Next, in order to study the merits of the transformations performed by the region scheduler, the speedups obtained by using transformed program dependence graphs were compared to the speedups obtained by using untransformed graphs. The speedups were computed by taking the ratio of the time spent on executing the sequential code on a pipelined machine and the time spent on executing the parallel code on the RLIW machine. The results of the experiments are presented in Table I. The first set of results (basic blocks) indicates the speedup obtained due to the parallelism detected only in individual basic blocks. The second set of results (untransformed regions) indicates the speedup obtained due to the parallelism detected in regions to which no transformations were applied by the region scheduler. The third

TABLE I
SPEED-UP DUE TO REGION SCHEDULING

	Basic Blocks	Untransformed Regions	Transformed Regions (δ_s=8)
TAYLOR1	2.41	2.68	4.06
TAYLOR2	1.96	2.15	3.60
EXACT	1.83	1.88	2.42
FFT	1.31	1.54	2.52
SORT	1.85	2.44	2.44
COLOR	1.16	1.64	1.64

set of results (transformed regions) indicates the speedup obtained due to the parallelism detected in regions to which transformations were applied by the region scheduler. In these experiments, no optimizations or transformations other than the transformations performed by the region scheduler were performed on the EPDG.

From the results it can be seen that the speedup obtained due to the parallelism detected in basic blocks is lower than that obtained due to parallelism detected in regions. The fact that a region is at least as large in size as a basic block accounts for this increase in parallelism for regions. Furthermore the results indicate that the application of transformations by the region scheduler caused a significant increase in the overall speedup achieved. For the programs TAYLOR1, TAYLOR2, EXACT, and FFT the parallelism detected by the region scheduler is limited by the size of the machine and the input to the program. Thus further improvement in performance can be obtained. No additional improvement in performance was obtained due to region scheduling for programs SORT and COLOR, for no regions in these programs had excess parallelism, and it was not possible to create excess parallelism by applying loop unrolling.

VII. Comparison of Trace and Region Scheduling

Region scheduling has a number of advantages over trace scheduling, a technique currently being used in a compiler to process fine grain parallelism for a long word instruction machine [7]. Trace scheduling uses the control flow graph representation of a program. It starts with innermost loop-free code and based upon predictions for the branch selections, chooses a path of basic blocks, called a trace, with the highest probability of execution. A single data dependence graph is constructed for the entire trace. Special edges are added to this graph to prevent the scheduler from performing illegal code motions across basic blocks. Next the scheduler treats this graph as a single basic block and generates a parallel instruction schedule. After scheduling the parallel code, the code motions across the jumps in the trace are examined, and compensation code is added at the entries and exits of the trace to preserve the semantics of the program. This is similar to updates involving copying of code when certain transformations are performed during region scheduling. The above process is repeated by selecting the next most likely trace until the entire program has been processed. In trace scheduling, the generation of traces and code generation are done together. In other words, construction of a new trace does not start until code has been generated for the trace last constructed. Through its ability to reorder code across jumps, trace scheduling allows detection of parallelism across basic blocks.

One of the advantages of region scheduling is that is uses a common program structure for traditional optimizations, parallelization, and scheduling. Many optimizations can be performed more efficiently than if a control flow graph, inherent in trace scheduling, is used [5]. The transformations for increasing parallelism in structured programs [1] and traditional optimizations can be applied to the EPDG. In addition, the transformations described in the last section, valid for both structured and unstructured programs, can be applied.

The transformations performed by the region scheduler are not heuristical in nature and thus, unlike trace scheduling, the performance of region scheduling does not vary with the structure of the input program. Both region and trace scheduling use reordering of code to generate a schedule of instructions that enables parallel execution. In trace scheduling, transformations are applied based on the statistical predication of execution frequency. In region scheduling, transformations are driven by the detection of parallel opportunities. A region scheduler can exploit at least as much parallelism present in the program as the trace scheduler, for any reordering that can be performed by a trace scheduler can be performed by a region scheduler. The transformations performed by the trace scheduler on the control flow graph enable it to move statement nodes across joins and splits in the control flow graph. The region scheduler can perform the same transformations through forward/backward code motion. In addition, the region scheduler is capable of moving entire subgraphs, for example, an if-statement, from one region to another. The region scheduler can apply more transformations, namely the loop transformations and region collapsing and merging transformations, thus enabling the detection of more parallelism.

Consider Figs. 12(a)–(c), which display transformations performed by the trace scheduler and the equivalent transformations as performed by the region scheduler using the EPDG. Figs. 12(a) and (b) show how the trace scheduler achieves reordering of operations B and C. In Fig. 12(a) it does so by moving operation C above a join in the control flow graph and in Fig. 12(b) by moving operation C past a conditional split. The region scheduler achieves the same effect by applying $\tau^{f_2}_{\text{move}}(R_1 \rightarrow R_2)$ and including B and C in the same region, R_2. The trace scheduler can also move operations above joins involving unconditional branches. The region scheduler can achieve the same effect using the transformation $\tau^{f_1}_{\text{move}}$. The code motion in Fig. 12(c) shows how the two schedulers achieve reordering of operation P_A and C. The region scheduler in this case applies transformation $\tau^{2}_{b}(R_2 \rightarrow R_1)$.

The transformation in Fig. 12(c) is applied by the trace scheduler when the probability of the predicate A being true is higher than that of it being false. This causes the

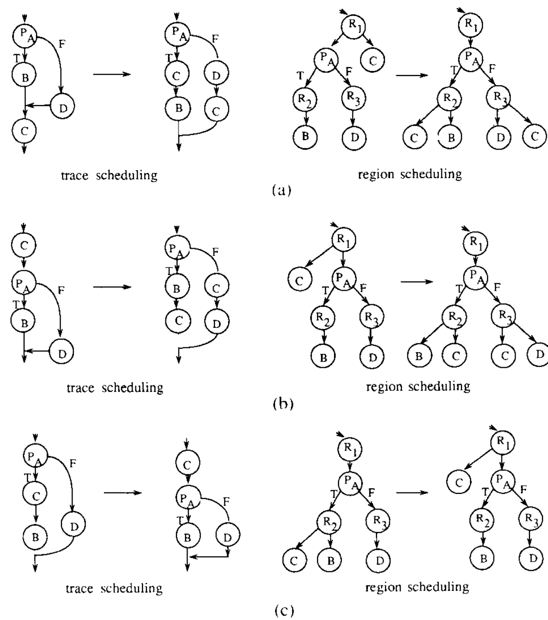


Fig. 12. Trace scheduling versus region scheduling.

trace scheduler to perform poorly if the probability of conditionals being true or false is equal, because trace scheduling generates a faster schedule for the trace more likely to be executed at the expense of the other. In region scheduling this drawback is avoided because the transformations are directed towards increasing the parallelism in the regions that do not have enough at the expense of the regions that contain excess parallelism. The transformation in Fig. 12(c) is only applied by the region scheduler when region R_1 has insufficient parallelism to utilize the system resources.

In trace scheduling the generation of traces and code generation are done together while in region scheduling the code generation is done after all the transformations have been performed on the regions. A disadvantage of the approach taken by the trace scheduler is that it does not take into account the system's processing capability when constructing traces. Thus some traces chosen by a trace scheduler may have more parallelism than the system can exploit while others may not have enough. Better code would be generated if the traces were modified so that the excess parallelism of one trace is distributed among the traces that do not have enough parallelism. Another disadvantage of constructing traces and generating code together is that the trace scheduler might duplicate code without actually utilizing the parallelism that the duplication might create. The example in Fig. 13 demonstrates this. Trace T_3, which is created by duplicating F, is treated as a separate trace instead of being merged with T_1. This happens because T_1 was the first trace to be chosen and the code for it had been generated before T_3 was created. It is likely that better code would be generated if traces T_1 and T_3 are merged.

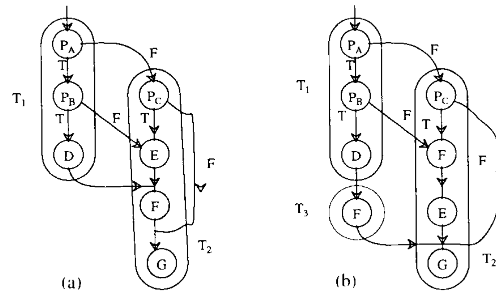


Fig. 13. Duplication without utilizing parallelism.

Sometimes in trace scheduling the trace prematurely stops growing, thus reducing the opportunities for the detection of parallelism [4]. This happens because the trace scheduler constructs traces based on the probabilities of execution of statements. This problem does not exist in a region scheduler, for regions are not constructed by any heuristic but are defined by the control structure of the program. In the example [4] in Fig. 14, trace T_1 consists of only A, P_1 and P_2, although it could have been longer. This situation arises when the execution estimate of branch from P_2 to P_3 is less than the execution estimate of branch from P_2 to F and the most likely path leading to F is through P_3. However in the EPDG for the same piece of code, region R_1 consists of A, P_1, P_2, P_3, and G and hence parallelism among them can be utilized.

To exploit the parallelism present among the statements before and after a loop, it should be possible to include these statements in the same trace. Thus a trace must be able to extend across loop boundaries. As described by Fisher [6] this requires additional analysis to determine the order in which the loops should be processed. In region scheduling this is not needed as the statements before and after the loop are already in the same region. The example in Fig. 15 demonstrates this. Statements A and C are in the same region R_1 and thus the region scheduler is able to exploit the parallelism between them. This is not done in trace scheduling because A and C are not included in the same trace.

Both region scheduling and trace scheduling may increase the size of the program exponentially. However, in the worst case region scheduling does not increase the size of the code as much as trace scheduling and, importantly, maintains control over whether or not to duplicate code. In both scheduling schemes, code is duplicated when operations are moved across branches and the worst case arises when the following sequence of if-statements occurs:

if P_1 then A_1 else B_1
if P_2 then A_2 else B_2
if P_n then A_n else B_n

In trace scheduling, code explosion may increase the code size to $O(n^n)$, where n is the number of if-statements [4]. However, region scheduling, in the worst case,

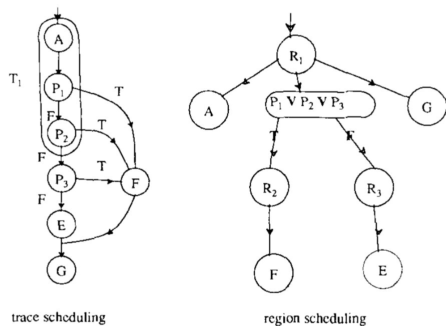


Fig. 14. Premature termination of a trace.

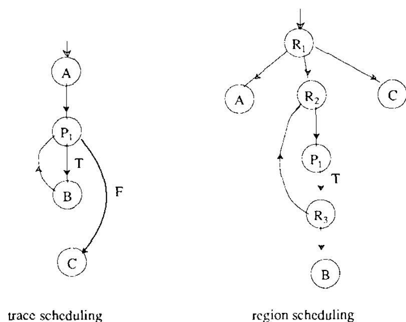


Fig. 15. Parallelism across loops.

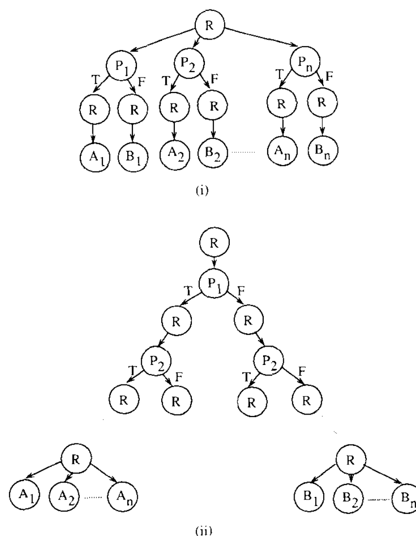
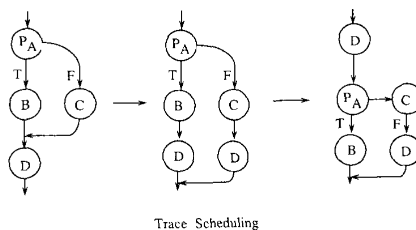


Fig. 16. Code duplication.



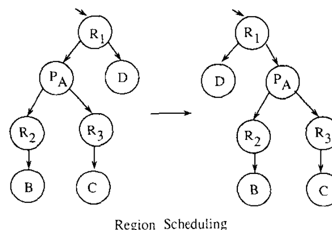


Fig. 17. Redundant code.

will increase the code size to $O(n2^n)$ as shown in Fig. 16. Code duplication can be easily avoided in region scheduling as the extent of duplication that will occur due to a transformation is known before it is applied, unlike trace scheduling where it is known only after a schedule for a trace has been generated. Thus, in region scheduling duplication of code can be controlled by avoiding duplication in parts of the program that are less likely to be executed.

When duplication of code is performed, redundant copies of code may be created. A redundant copy of code is one which if deleted does not affect the results of execution of a program. This will make the generated code less than optimal. When the trace scheduler moves code across both a join and a split, redundant code is generated. In the example in Fig. 17, the trace scheduler in the process of reordering P_A and D creates a redundant copy of D. On the other hand the region scheduler does not need to duplicate code to reorder P_A and D as they are already in the same region. Thus, the region scheduler will not generate redundant code when a statement node is moved across both a join and a split. For structured programs, no redundant code is ever generated by the region scheduler during the local transformation phase.

Lastly, the application of some transformations in region scheduling reduces the number of branches present in the program. This results in the improvement in the performance of pipelined machines.

VIII. Conclusion

A technique for detecting and redistributing fine grain parallelism is presented in this paper. The technique uses an extension of the PDG and thus inherits the advantages of using this representation, such as the use of a common structure for efficiently applying both traditional optimizations and vectorizing transformations and incrementally updating data dependency information.

The extension permits the use of the same representation in also detecting and redistributing fine grain parallelism and code generation. The redistribution is driven by an estimate of the parallelism in each region. The use of an intermediate representation of a program makes it applicable to programs written in a wide range of languages, and both structured and unstructured programs

can be represented. In addition, it is architecture independent in that any architecture that has fine grain parallel capabilities can use the technique.

Experiments performed demonstrate the effectiveness of the transformations. Although region scheduling employs the EPDG which is more expensive to construct than a control flow graph, its advantages, especially its applicability to a wide range of programs and increased power, make it an attractive alternative to trace scheduling. Although the use of the EPDG in this work has been oriented to the detection and distribution of fine grain parallelism, future work will consider the use of the EPDG for detection of coarse grain units for multiprocessing.

Acknowledgment

We are grateful to J. Ferrante and K. Ottenstein for their comments and suggestions on this work. We also thank the referees for their suggestions in improving this paper.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools.* Reading, MA: Addison-Wesley, 1986.

[2] J. Allen and K. Kennedy, "Automatic loop interchange," in *Proc. SIGPLAN Symp. Compiler Construction*, vol. 19, no. 6, 1984, pp. 233-246.

[3] M. Burke, R. Cytron, J. Ferrante, W. Hsieh, V. Sarkar, and D. Shields, "Automatic discovery of parallelism: A tool and an experiment," in *Proc. ACM/SIGPLAN Symp. Parallel Programming: Experience with Applications, Languages and Systems*, July, 1988, pp. 77-84.

[4] J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures.* Cambridge, MA: MIT Press, 1986.

[5] J. Ferrante, K. Ottenstein, and J. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319-349, July 1987.

[6] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. Comput.*, vol. C-30, pp. 478-490, July 1981.

[7] ——, "VLIW architectures: Supercomputing via overlapped execution," in *Proc. Second Int. Conf. Supercomputing*, vol. I, May 1987, pp. 353-361.

[8] R. Gupta, "A reconfigurable LIW architecture and its compiler," Ph.D. dissertation, Dep. Comput. Sci., Univ. Pittsburgh, Tech. Rep. 87-3, Aug. 1987.

[9] R. Gupta and M. L. Soffa, "A reconfigurable LIW architecture," in *Proc. Int. Conf. Parallel Processing*, Aug. 1987, pp. 893-900.

[10] J. Hennessy and T. Gross, "Postpass code optimization of pipeline constraints," *ACM Trans. Program. Lang. Syst.*, vol. 3, no. 5, pp. 422-448, 1983.

[11] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence graphs and compiler optimizations," in *Proc. 8th Annu. ACM Symp. Principles of Programming Languages*, 1981, pp. 207-218.

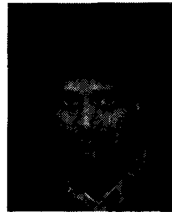
[12] D. J. Kuck, *The Structure of Computers and Computations*, vol. 1. New York: Wiley, 1978.

[13] D. A. Padua, D. J. Kuck, and D. Lawrie, "High-speed multiprocessors and compilation techniques," *IEEE Trans. Comput.*, vol. C-29, no. 9, pp. 763-776, 1980.

[14] L. L. Pollock and M. L. Soffa, "A incremental version of iterative data flow analysis," *IEEE Trans. Software Eng.*, vol. 15, no. 12, Dec. 1989.

[15] V. Sarkar and J. Hennessy, "Compile time partitioning and scheduling of parallel programs," in *Proc. Symp. Compiler Construction*, 1986, pp. 17-26.

[16] R. Sites, "Instruction ordering for the Cray-1 computer," Dep. EECS, Univ. California at San Diego, Tech. Rep. CS-023, July 1978.



Rajiv 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, Pittsburgh, PA, in 1987.

Since then he has been a senior member of Research Staff in the Computer Architecture and Programming Systems group at Philips Laboratories, Briarcliff Manor, NY. His primary research interests include compilation techniques for parallel systems, parallel architectures, and implementation of programming languages.

Dr. Gupta is a member of the Association for Computing Machinery, Sigplan, and the IEEE Computer Society.

Mary Lou Soffa received the Ph.D. degree in computer science from the University of Pittsburgh, Pittsburgh, PA, in 1977.

She has been on the faculty at the University of Pittsburgh since 1977 and is currently an Associate Professor in the Department of Computer Science. Her main areas of research interest are compiling techniques for parallel systems, incremental compilation, implementation of programming languages, and software tools.

Dr. Soffa serves on the Editorial Advisory Board for *Computer Languages* and is a member of the Association for Computing Machinery, Sigplan, Sigsoft, and the IEEE Computer Society.