Interprocedural Transformations for Parallel Code Generation

Mary Hall
Ken Kennedy
Kathryn McKinley

CRPC-TR91149-S
April 1991

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Appears in *The Proceedings of Supercomputing '91.*

1

Interprocedural Transformations for Parallel Code Generation

Mary W. Hall Ken Kennedy Kathryn S. McKinley
Department of Computer Science, Rice University, Houston, TX 77251-1892

Abstract

We present a new approach that enables compiler optimization of procedure calls and loop nests containing procedure calls. We introduce two interprocedural transformations that move loops across procedure boundaries, exposing them to traditional optimizations on loop nests. These transformations are incorporated into a code generation algorithm for a shared-memory multiprocessor. The code generator relies on a machine model to estimate the expected benefits of loop parallelization and parallelism-enhancing transformations. Several transformation strategies are explored and one that minimizes total execution time is selected. Efficient support of this strategy is provided by an existing interprocedural compilation system. We demonstrate the potential of these techniques by applying this code generation strategy to two scientific applications programs.

1 Introduction

Modern computer architectures, such as pipelined, superscalar, VLIW and multiprocessor machines, demand sophisticated compilers to exploit their performance potentials. To expose parallelism and computation for these architectures, the compiler must consider a statement in light of its surrounding context. Loops provide a proven source of both context and parallelism. Loops with significant amounts of computation are prime candidates for compilers seeking to make effective utilization of the available resources. Given that increased modularity is encouraged to manage program computation and complexity, it is natural to expect that programs will contain many procedure calls and procedure calls in loops, and the ambitious compiler will want to optimize them.

Unfortunately, most conventional compiling systems abandon parallelizing optimizations on loops containing procedure calls. Two existing compilation technologies are used to overcome this problem: interprocedural analysis and interprocedural transformation.

Interprocedural analysis applies data-flow analysis techniques across procedure boundaries to enhance the effectiveness of dependence testing. A sophisticated form of interprocedural analysis, called *regular section analysis*, makes it possible to parallelize loops with calls by determining whether the side effects to arrays as a result of each call are limited to nonintersecting subarrays on different loop iterations [12, 20].

Interprocedural transformation is the process of moving code across procedure boundaries, either as an optimization or to enable other optimizations. The most common form of interprocedural transformation is *procedure inlining*. Inlining substitutes the body of a called procedure for the procedure call and optimizes it as a part of the calling procedure.

Even though regular section analysis and inlining are frequently successful, each of these methods has its limitations [20, 23]. Compilation time and space considerations require that regular section analysis *summarize* array side effects. In general, summary analysis for loop parallelization is less precise than the analysis of inlined code. On the other hand, inlining can yield an explosion in code size which may disastrously increase compile time and seriously inhibit separate compilation [13]. Furthermore, inlining may cause a loss of precision in dependence analysis due to the complexity of subscripts that result from array parameter reshapes. For example, when the dimension size of a formal array parameter is also passed as a parameter, translating references of the formal to the actual can introduce multiplications of unknown symbolic values into subscript expressions. This situation occurs when inlining is used on the SPEC Benchmark program matrix300 [8].

In this paper, a hybrid approach is developed that overcomes some of these limitations. We introduce a pair of new interprocedural transformations: *loop embedding*, which pushes a loop header into a procedure called within the loop, and *loop extraction*, which extracts the outermost loop from a procedure body into the calling procedure. These transformations expose such loops to intraprocedural optimizations. In this paper, the intraprocedural optimizations considered are loop fusion, loop interchange and loop distribution. However, many other transformations that require loop nests will also benefit from embedding and extraction. Some examples are loop skewing [36] and memory hierarchy optimizations such as unroll and jam [10].

As a motivating example, consider the Fortran code in Example 1(a). The `J` loop in subroutine `S` may safely be made parallel, but the outer `I` loop in subroutine `P` may not be. However, the amount of computation in the `J` loop is small relative to the `I` loop and may not be sufficient to make parallelization profitable. If the `I` loop is *embedded* into subroutine `S` as shown in (b), the

*This research was supported by the Center for Research on Parallel Computation, a National Science Foundation Science and Technology Center, by IBM Corporation, the state of Texas and by a DARPA/NASA Research Assistantship in Parallel Processing, administered by the Institute for Advanced Computer Studies, University of Maryland.

```
SUBROUTINE P
  REAL A(N,N)
  INTEGER I
  DO I = 1, 100
    CALL S(A,I)
  ENDDO
SUBROUTINE S(F,I)
  REAL F(N,N)
  INTEGER I,J
  DO J = 1,3
     F(J,I) = F(J,I-1) + 10
  ENDDO
```

(a) before transformation

```
SUBROUTINE P
  REAL A(N,N)


  CALL S(A)

SUBROUTINE S(F)
  REAL F(N,N)
  INTEGER I,J
  DO I = 1, 100
    DO J = 1, 3
      F(J,I) = F(J,I-1) + 10
    ENDDO
  ENDDO
```

(b) loop embedding

```
SUBROUTINE P
  REAL A(N,N)


  CALL S(A)

SUBROUTINE S(F)
  REAL F(N,N)
  INTEGER I,J
  PARDO J = 1, 3
    DO I = 1, 100
      F(J,I) = F(J,I-1) + 10
    ENDDO
  ENDPARDO
```

(c) loop interchange

Example 1:

inner and outer loops may be interchanged as shown in (c). The resulting parallel outer `J` loop now contains plenty of computation. As an added benefit, procedure call overhead has been reduced.

Loop embedding and loop extraction provide many of the optimization opportunities of inlining without its significant costs. Code growth of individual procedures is nominal, so compilation time is not seriously affected. Overall program growth is also moderate because multiple callers may invoke the same optimized procedure body. In addition, the compilation dependences among procedures are reduced since the compiler controls the small amount of code movement across procedures and can easily determine if an editing change of one procedure invalidates other procedures.

Our approach to interprocedural optimization is fundamentally different from previous research in that the application of interprocedural transformations is restricted to cases where it is determined to be profitable. This strategy, called *goal-directed interprocedural optimization*, avoids the costs of interprocedural optimization when it is not necessary[8]. Interprocedural transformations are applied as dictated by a code generation algorithm that explores possible transformations, selecting a choice that minimizes total execution time. Estimates of execution time are provided by a machine model which takes into account the overhead of parallelization. The code generator is part of an interprocedural compilation system that efficiently supports interprocedural analysis and optimization by retaining separate compilation of procedures.

The remainder of this paper is organized into five major sections, related work, and conclusions. Section 2 provides the technical background for the rest of the paper. In Section 3, a compilation system is described which is powerful enough to support interprocedural optimization but also retains the advantages of a separate compilation system. Section 4 explains the interprocedural and intraprocedural transformations in more detail, and Section 5 presents a code generation algorithm that uses these to parallelize programs for a shared-memory multiprocessor. Section 6 describes an experiment where this approach was applied to the Perfect Benchmark programs `spec77` and `ocean`.

2 Technical Background

2.1 Dependence Analysis

Dependence analysis and testing have been widely researched, and in this paper a working knowledge of these is assumed [3, 7, 9, 17, 18, 27, 37]. In particular, the reader should be familiar with dependence graphs, where dependence edges are characterized with such information as dependence type and hybrid direction/distance vectors [25]. The dependence graph specifies a conservative approximation of the partial order of memory accesses necessary to preserve the semantics of a program. The *safe* application of program transformations is based on preserving this partial order.

2.2 Augmented Call Graph

The program representation for interprocedural transformations requires an *augmented call graph* to describe the calling relationship among procedures and specify loop nests. The code generation algorithm considers loops containing procedure calls and loops adjacent to procedure calls. For this purpose, the program's call graph, which contains the usual *procedure nodes* and *call edges*, is augmented to include special *loop nodes* and *nesting edges*. If a procedure p contains a loop l, there will be a nesting edge from the procedure node representing p to the loop node representing l. If a loop l contains a call to a procedure p, there will be a nesting edge from l to p. Any inner loops are also represented by loop nodes and are children of their outer loop. The outermost loop of each routine is marked *enclosing* if all the other statements in the procedure fall inside the loop. Figure 1(a) shows the augmented call graph for the program from Example 1.

2.3 Regular Section Analysis

A regular section describes the side effects to the substructures of an array. Sections represent a restricted set of the most commonly occurring array access patterns; single elements, rows, columns, grids and their higher dimensional analogs. This restriction on the shapes assists in making the implementation

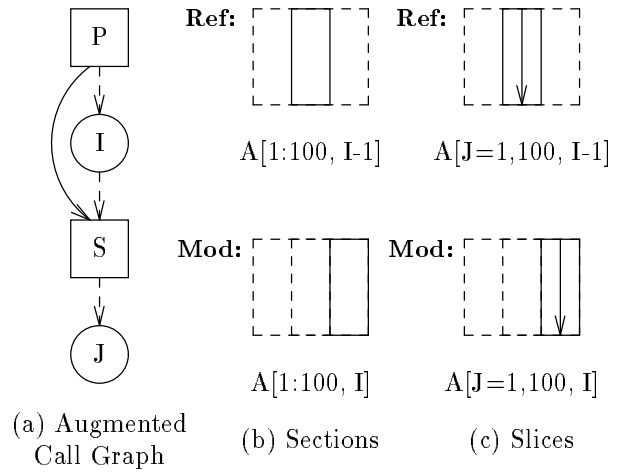

Figure 1:

efficient [20]. The representation of the dimensions of a particular array variable may take one of three forms: (1) an invocation invariant expression, representing a single element; (2) a range consisting of a lower bound, an upper bound and a step size; or (3) the special element $\perp$, signifying that all of this dimension may be affected. Sections are separated into modified and referenced sets. The sections for Example 1 are shown in Figure 1(b).

By using sections, the problem of locating dependences on procedure calls is simplified to the problem of finding dependences on ordinary statements. The modified and referenced subsections for the call appear to the dependence analyzer like the left- and right-hand sides of an assignment, respectively. For single-element subsections, dependence testing is the same as it would be for any other variable access. For subsections that contain one or more dimensions with ranges, the dependence analyzer simulates DO loops for each of the range dimensions, with the lower bound, upper bound and step size of the loop corresponding to those of the range. Sections are necessarily an approximation of actual accesses. To assist conservative dependence testing, they are marked exact and inexact to indicate whether they are an approximation.

Regular sections enable dependence analysis to determine if loops containing calls are parallel. Sections are also currently used to determine the safety of intraprocedural transformations on a loop nest containing calls. In this paper, sections are extended to enable the code generator to determine the safety of interprocedural transformations. We introduce an annotation to a section, called a *slice*. Slices resemble *data access descriptors*, but they are not as detailed [5]. A slice identifies the section of an array accessed and the order of that access in terms of a particular loop's index expression. Symbolic slices are stored only for the outermost loop of a procedure. They are also marked as exact or inexact. Figure 1(c) illustrates the slice annotations for the program in Example 1.

3 Support for Interprocedural Optimization

In this section, we present the compilation system of the ParaScope Programming Environment [11, 14]. This system was designed for the efficient support of interprocedural analysis and optimization. The tools in ParaScope cooperate to enable the compilation system to perform interprocedural analysis without direct examination of source code. This information is then used in code generation to make decisions about interprocedural optimizations. The code generator only examines the dependence graph for the procedure currently being compiled, not the graph for the entire program. In addition, ParaScope employs *recompilation analysis* after program changes to minimize program reanalysis [15].

3.1 The ParaScope Compilation System

Interprocedural analysis in the ParaScope compilation system consists of two principal phases. The first takes place prior to compilation. At the end of each editing session, the immediate interprocedural effects of a procedure are determined and stored. For example, this information includes the array sections that are locally modified and referenced in the procedure. The procedure's calling interface is also determined in this phase. It includes descriptions of the calls and loops in the procedure and their relative positions. In this way, the information needed from each module of source code is available at all times and need not be derived on every compilation.

Interprocedural optimization is orchestrated by the *program compiler*, a tool that manages and provides information about the whole program [14, 19]. The program compiler begins by building the augmented call graph described in Section 2.2. The program compiler then traverses the augmented call graph, performing interprocedural analysis, and subsequently, code generation. Conceptually, program compilation consists of three principal phases: (1) interprocedural analysis, (2) dependence analysis, and (3) planning and code generation.

Interprocedural analysis. The program compiler calculates interprocedural information over the augmented call graph. First, the information collected during editing is recovered from the database and associated with the appropriate nodes and edges in the call graph. This information is then propagated in a top-down or bottom-up pass over the nodes in the call graph, depending on the interprocedural problem. Section analysis is performed at this time. Interprocedural constant propagation and symbolic analysis are also performed, as these greatly increase the precision of subsequent dependence analysis.

Dependence analysis. Interprocedural information is then made available to dependence analysis, which is performed separately for each procedure. Dependence analysis results in a dependence graph. Edges in the dependence graph connect statements that form the source and sink of a dependence. If the source or sink of a dependence is a call site, a sec-

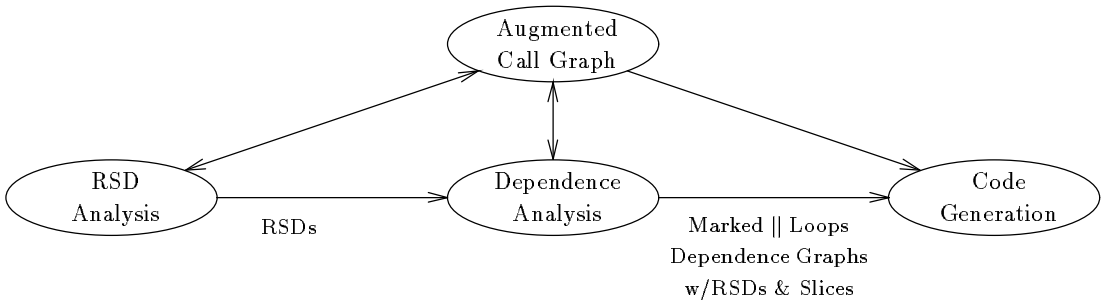

Figure 2: Flow of information for interprocedural transformations.

tion annotates it. The section may more accurately describe the portion of the array involved in the dependence. Dependence analysis also distinguishes parallel loops in the augmented call graph. Dependence analysis is separated from code generation for an important reason; it provides the code generator knowledge about each procedure without reexamining their source or dependence graph.

Planning and Code Generation. The final phase of the program compiler determines where interprocedural optimization is profitable. When more than one option for interprocedural transformation exists, it selects the most profitable option. Planning is important to interprocedural optimization since unnecessary optimizations may lead to significant compile-time costs without any execution-time benefit. To determine the profitability of transformations requires a machine model. To determine the safety of transformations, the dependence graph and sections are sufficient. Once profitable transformations are located, they are applied and parallelism is introduced in the transformed program.

The relationship among the compilation phases is depicted in Figure 2. Each step adds annotations to the call graph that are used by the next phase. Following program transformation, each procedure is separately compiled. Interprocedural information for a procedure is provided to the compiler to enhance *intraprocedural* optimization.

3.2 Recompilation Analysis

A unique part of the ParaScope compilation system is its recompilation analysis, which avoids unnecessary recompilation after editing changes to the program. Recompilation analysis tests that interprocedural facts used to optimize a procedure have not been invalidated by editing changes [15]. To extend recompilation analysis for interprocedural transformations, a few additions are needed. When an interprocedural transformation is performed, a description of the interprocedural transformations annotates the nodes and edges in the augmented call graph. On subsequent compilations, this information indicates to the program compiler that the same tests used initially to determine the safety of the transformations should be reapplied.

To determine if interprocedural transformations are still safe, the new and old sections are first compared, in most cases avoiding examination of the dependence graph. This means that dependence analysis is only applied to procedures where it is no longer valid, allowing separate compilation to be preserved. The recompilation process after interprocedural transformations have been applied is described in more detail elsewhere [19].

4 Interprocedural Transformation

We introduce two new interprocedural transformations, loop extraction and loop embedding. These expose the loop structure to optimization without incurring the costs of inlining. The movement of a single loop header is detailed below. Moving additional statements that precede or are enclosed by a loop is a straightforward generalization of these two transformations and for simplicity is not described. This section also describes the additional information needed to perform the applicability and safety tests for loop fusion and loop interchange across call boundaries. All of these are used in our code generation algorithm. The code generation algorithm also uses loop distribution, but does not apply it across call boundaries. Therefore, it may be performed with no additional information. Loop distribution is discussed in detail in Section 5.2.

4.1 Loop Extraction

Loop extraction moves an enclosing loop of a procedure p outward into one of its callers. This optimization may be thought of as partial inlining. The new version of p no longer contains the loop. The caller now contains a new loop header surrounding the call to p. The index variable of the loop, originally a local in p, becomes a formal parameter and is passed at the call. The calling procedure creates a new variable to serve as the loop index, avoiding name conflicts. It is always safe to extract an outer enclosing loop from a procedure. Example 2(a) contains a loop with two calls to procedure `S` and (b) contains the result after loop extraction. Note that (b) has an additional variable declaration for the loop index `J` in `P`. It is included in the actual parameter list for `S`. In this example, the `J` loop may now be fused and interchanged to improve performance.

4.2 Loop Embedding

Loop embedding moves a loop that contains a procedure call into the called procedure and is the dual of loop extraction. The new version of the called procedure requires a new local variable for the loop's index variable. If a name conflict exists, a new name for the loop's index variable must be created. This transformation is illustrated in Example 1.

```
SUBROUTINE P(A)
  REAL A(N,N), B(N,N)
  INTEGER I

  DO I = 1, 3
    CALL S(A,I)
    CALL S(B,I)
  ENDDO

SUBROUTINE S(F,I)
  REAL F(N,N)
  INTEGER I,J
  DO J = 1,100
    F(J,I) = F(J,I) + 10
  ENDDO
```

(a) before transformation

```
SUBROUTINE P(A)
  REAL A(N,N), B(N,N)
  INTEGER I,J
  DO I = 1, 3
    DO J = 1, 100
      CALL S(A,I,J)
    ENDDO
    DO J = 1, 100
      CALL S(B,I,J)
    ENDDO
  ENDDO
SUBROUTINE S(F,I,J)
  REAL F(N,N)
  INTEGER I,J

  F(J,I) = F(J,I) + 10
```

(b) loop extraction

Example 2:

If the index variable of the loop to be embedded appears in an actual parameter in the call, this parameter is no longer correctly defined. To remedy this problem, the formals that depend on it must be assigned and computed in the newly embedded loop. In the simplest case, an index variable i is passed to a formal f. Here, f should be assigned i on every iteration of the embedded loop, prior to the rest of the loop body.

If an actual is an array reference whose subscript expression contains the loop index variable, the actual passed at the call becomes simply the array name. In the called procedure, the original subscript expression for each dimension of the actual is added to the subscript expression for the corresponding dimension of the formal at each reference to the formal. If the array parameter is reshaped across the call, this translation is more complicated. The array formal is replaced by a new array with the same shape as the actual. The references to the variable are translated by linearizing the formal's subscript expressions and then converting to the dimensions of the new array[9]. Finally, the subscript expressions for each dimension of the actual are added to those for the translated reference. This method is also the one that is used in inlining.

Procedure Cloning

Procedures optimized with embedding or extraction may have multiple callers, and an optimization valid for one caller may not be valid for another. To avoid significant code growth, multiple callers should share the same version of the optimized procedure whenever possible. This technique of generating multiple copies of a procedure and tailoring the copies to their calling environments is called procedure cloning [14].

Dependence Updates

Because our code generator only applies loop extraction and loop embedding after safety and profitability are ensured, an update of local dependence information is not necessary. However, if further optimization is desired, updating the dependence information is straightforward.

4.3 Loop Fusion

Loop fusion places the bodies of two adjacent loops with the same number of iterations into a single loop [1]. When several procedure calls appear contiguously or loops and calls are adjacent, it may be possible to extract the outer loop from the called procedure(s), exposing loops for fusion and further optimization. In the algorithm *checkFusion*, we consider fusion for an ordered set $S = \{s_1, \ldots, s_p\}$, where s_i is either a call or a loop. There cannot be any intervening statements between s_i and s_{i+1} and each call must contain an enclosing loop which is being considered for fusion.

Fusion is safe for two loops l_1 and l_2 if it does not result in values flowing from the statements in l_2 back into the statements in l_1 in the resultant loop and vice versa. The simple test for safety performs dependence testing on the loop bodies as if they were in a single loop. Each forward dependence originally between l_1 and l_2 is tested. Fusion is unsafe if any dependences are reversed, becoming backward loop-carried dependences in the fused loop.

This test requires the inspection of the dependence source and sink variable references in l_1 and l_2. If one or more of the loops is inside a call, the variable references are represented instead as the modified and referenced sections for the call. The slices that annotate the sections correspond to the loops being considered for fusion and are tested identically to variable references (see Section 2.3). Unfortunately, while variable references are always exact, a section and its slice are not. If the slice is not exact, fusion is conservatively assumed to be unsafe. To be more precise would require the inspection of the dependence graphs for each called procedure, possibly a significant overhead.

checkFusion (S)
/* Input: $S = \{s_1, \ldots, s_p\}$, s_i is a call or a loop */
/* s_i is adjacent to s_{i+1} */
/* Output: returns true if fusion is safe $\forall$ s_i */
$\mathcal{F} = \{s_1\}$
for $i = 2$ to n
  let l_i = the loop header of s_i
  if the number of iterations of l_i differ from $\mathcal{F}$ **then**
    return false
  for each forward dependence ($src_{\mathcal{F}}$, $sink_{s_i}$)
    if $src_{\mathcal{F}}$ or $sink_{s_i}$ is not exact **then**
      return false
    if ($src_{\mathcal{F}}$, $sink_{s_i}$) becomes
      backward loop-carried **then**
      return false
  endfor
  $\mathcal{F} = \mathcal{F} \cup \{s_i\}$
endfor
return true

4.4 Loop Interchange

Loop interchange of two nested loops exchanges the loop headers, changing the order in which the itera-

tion space is traversed. It is used to introduce parallelism or to adjust granularity of parallelism. In particular, when a loop containing calls is not parallel or parallelizing the loop is not profitable, it may be possible to move parallel loops in the called procedures outward using loop interchange as in Examples 1 and 2. The safety of loop interchange may be determined by inspecting the distance/direction vector to ensure that no existing dependence is reversed after interchange [3, 37].

Our algorithm considers loop interchange only when a perfect nest can be created via loop extraction, embedding, fusion, and distribution. If a loop contains more than one call, it may be possible to fuse the outer enclosing loops of calls to create a perfect nest. Even if there are multiple statements and calls, it may be possible to use loop distribution to create a perfect nest. If a perfect nest may be safely created, testing the safety of interchange simply requires inspection of the direction vectors and slices for dependences between calls or statements in the nest.

5 Interprocedural Parallel Code Generation

In this section we present an algorithm for the interprocedural parallel code generation problem. This algorithm moves loops across procedure boundaries when other transformations such as loop fusion, interchange, and distribution may be applied to the resulting loop nests to introduce or improve single-level loop parallelism. The goal of this algorithm is to only apply transformations which are proven to minimize execution time for a particular code segment. To determine the minimum execution time of a code segment, a simple machine model is used. This model includes the cost of arithmetic and conditional statements as well as operations such as parallel loops, sequential loops, and procedure call overhead. Both Polychronopoulos and Sarkar have used similar machine models in their research [33, 34].

5.1 Machine Model and Performance Estimation

A cost model is needed to compare the costs of various execution options. First, a method for estimating the cost of executing a sequential loop is presented. Consider the following perfect loop nest, where ub_1, ..., ub_n are constants and B is the loop body.

```
DO i1 = 1, ub1
  ...
  DO in = 1, ubn
    B
  ENDDO
  ...
ENDDO
```

In order to estimate the cost of running this loop on a single processor, a method for estimating the running time of the loop body is needed. If B consists of straight-line code, simply sum the time to execute each statement in the sequence. To handle control flow, we assume a probability for each branch and compute the weighted mean of the branches. Once the sequential running time of the loop body $t(B)$ is computed, then the running time for the inner loop is given by the formula:

$$ub_n(t(B) + o),$$

where o is the sequential loop overhead. The running time for the entire loop nest is then given by the following:

$$ub_1(\ldots(ub_n(t(B) + o)\ldots) + o).$$

In order to estimate the running time of a parallel loop, we need to take into account any overhead introduced by the parallel loop. Our experiments on uniform shared-memory machines indicate that this overhead consists of a fixed cost c_s of starting the parallel execution and a cost c_f of forking and synchronizing each parallel process. If there are P parallel processors, an estimate of the cost of executing the inner loop of the above example in parallel is given by the equation

$$c_s + c_f P + \left\lceil \frac{ub_n}{P} \right\rceil (t(B) + o) \ .$$

This formula assumes that the iterations are divided into nearly equal blocks at startup time and the overhead of an iteration o remains the same. Given a perfect loop nest where just one loop is being considered for parallel execution, these two formulae may be generalized to compute the expected sequential and parallel execution time. If the parallel execution time is less than the sequential execution time, it is profitable to run the loop in parallel.

To enable the parallel code generator to compare the costs of different transformation choices, we introduce the following cost function:

$cost(\mathcal{L}, how, B)$, where
$\mathcal{L} = \{l_1, \ldots l_n\}$, a perfect loop nest
how indicates whether l_n is parallel ($||$) or sequential
B = the loop body

The function *cost* estimates the running time of a loop nest $l_1, \ldots, l_n$, where the inner loop l_n is specified as either parallel or sequential, and all outer loops are sequential. The loop body B may contain any types of statements, including calls and inner loop nests.

5.2 Code Generation Algorithm

The goal of our interprocedural parallel code generation algorithm is to introduce effective loop parallelism for programs which contain procedure calls and loops. This algorithm applies the following transformations: loop fusion, loop interchange, loop distribution, loop embedding, loop extraction, and loop parallelization. These transformations are applied at call sites and for a loop nest containing call sites. The algorithm seeks a minimum cost single loop parallelization based on performance estimates.

Potential loop and call sequences that may benefit from these interprocedural transformations are adjacent procedure calls, loops adjacent to calls, and loop nests containing calls. To find candidates for interprocedural optimization, the augmented call graph is traversed in a top-down pass. If a candidate benefits

BestCost $(S, \mathcal{L})$

/* Input: a set of statements $S = \{s_1, \ldots, s_p\}$ in perfect loop nest $\mathcal{L} = \{l_1, \ldots l_n\}$ */
/* Output: a tuple $\langle \tau, \mathcal{T} \rangle$, where τ = the minimum execution time and */
/* $\mathcal{T}$ = the set of transformations that result in τ */

$\langle \tau, \mathcal{T} \rangle = \langle cost(\mathcal{L}, sequential, S), \emptyset \rangle$
if $(\mathcal{L} = \emptyset)$ **then**
  if ($checkFusion(S)$ & (fused loop l_f is $\|$)) **then**
    $\langle \tau, \mathcal{T} \rangle = min\ (\langle cost(l_f, \|, body(l_f)), \{\text{fuse, make } l_f\ \|\}\ \rangle, \langle \tau, \mathcal{T} \rangle)$
  return $\langle \tau, \mathcal{T} \rangle$
endif
for $(i = 1, n)$
  if $(l_i$ is $\|\ )$ **then**
    $\langle \tau, \mathcal{T} \rangle = min\ (\langle cost(\{l_1, \ldots, l_i\}, \|, body(l_i)), \{\text{make } l_i\ \|\}\ \rangle, \langle \tau, \mathcal{T} \rangle)$
    if $i \neq n$ **then return** $\langle \tau, \mathcal{T} \rangle$
  endif
endfor
if ($checkFusion(S)$) **then**
  if (fused loop l_f is $\|$) **then**
    if ($checkInterchange(l_n, l_f)$ & l_f is $\|$ after interchange) **then**
(1) $\langle \tau, \mathcal{T} \rangle = min(\langle cost(\{l_1, \ldots, l_{n-1}, l_f\}, \|, l_n * body(l_f)), \{\text{fuse, interchange, make } l_f\ \|\}\rangle, \langle \tau, \mathcal{T} \rangle)$
    else
(2) $\langle \tau, \mathcal{T} \rangle = min(\langle cost(\{l_1, \ldots, l_n, l_f\}, \|, body(l_f)), \{\text{fuse, make } l_f\ \|\}\ \rangle, \langle \tau, \mathcal{T} \rangle)$
  else if (l_n is $\neg\ \|$) & ($checkInterchange\ (l_n, l_f)$) & ($l_n$ $\|$ after interchange) **then**
(3) $\langle \tau, \mathcal{T} \rangle = min(\langle cost(\{l_1, \ldots, l_{n-1}, l_f, l_n\}, \|, body(l_f)), \{\text{fuse, interchange, make } l_n\ \|\}\rangle, \langle \tau, \mathcal{T} \rangle)$
  endif
return $\langle \tau, \mathcal{T} \rangle$

---

from interprocedural transformation, the transformations are performed and no further optimization of that call sequence is attempted. Additional candidates for optimization may be created by using judicious code motion and loop coalescing (combining nested loops into a single loop)[33].

BestCost Algorithm

BestCost considers $\mathcal{L} = \{l_1, \ldots, l_n\}$ a perfect loop nest with body $S = \{s_1, \ldots, s_p\}$, where l_n is the innermost loop and L may be the empty set $\emptyset$. S consists of at least one call and may also contain other statements such as loops, control flow, and assignments.

The *BestCost* algorithm makes use of loop parallelization, fusion, interchange, extraction, and embedding (loop distribution is excluded) to determine a tuple $\langle \tau, \mathcal{T} \rangle$, such that τ is the best execution time and $\mathcal{T}$ specifies the transformations needed to obtain this time. Unfortunately, finding the best ordering of a loop nest via loop interchange requires that all possible permutations ($n!$) be considered. Therefore to restrict the search space and simplify this presentation, *BestCost* only considers loop interchange of l_n the innermost nest and l_f the result of fusing S. However, opportunities to test various interchange strategies are pointed out in the text.

The sequential execution time is computed first ($\mathcal{T} = \emptyset$). If there is no surrounding loop nest ($\mathcal{L} = \emptyset$), S may be a group of adjacent calls and loops that can be fused. If fusion of all members of S is possible and produces a parallel loop, its execution time is computed and compared to the sequential cost using the function *min*. The function *min* assigns τ the minimum of the two times, and $\mathcal{T}$ the corresponding program transformation. If $\mathcal{L} \neq \emptyset$, other transformations are considered as follows.

First, the outermost parallel loop of $\mathcal{L}$ is sought and compared with the sequential time. If any of $l_1 \ldots l_{n-1}$ are parallel, *BestCost* returns. Loop interchange outward of any of these parallel loops could also be considered. Otherwise, if all of S fuses into l_f, three transformations on l_f and l_n are considered.

1. Interchanging a parallel l_f with l_n to make a parallel loop with increased granularity.
2. A parallel l_f in its current position.
3. Interchanging l_n and l_f to introduce inner loop parallelism.

Case 1 is illustrated in Examples 1 and 2. Further interchanging of l_f to enable a more outer loop to be parallel may also be tested here.

Embedding versus Extraction

To apply the set of transformations specified by $\langle \tau, \mathcal{T} \rangle$, the loops involved may need to be placed in the same routine. In particular, if $\mathcal{T}$ specifies interchange or fusion across a call then one of embedding or extraction must be applied. If there is only one call, then embedding loop l_n into the called procedure is preferable because it reduces procedure call overhead. If there is more than one call and $\mathcal{T}$ requires fusion, extraction from all the calls is performed. Fusion, inter-

change, and parallelization may then be performed on the transformed loops.

Loop Distribution

If *BestCost*($\mathcal{L}$, S) cannot introduce parallelism, then it may be possible to use loop distribution to do so. Loop distribution seeks parallelism by separating independent parallel and sequential statements in $\mathcal{L}$. For example, loop distribution may create loop nests of adjacent calls and loops which *BestCost* can optimize.

Ordered Partitions. Loop distribution is safe if the *partition* of statements into new loops preserves all of the original dependences [24, 32]. Dependences are preserved if any statements involved in a cycle of dependences, a *recurrence*, are placed in the same loop (partition). The dependences between the partitions then form an acyclic graph that can always be ordered using topological sort [3, 28].

By first choosing a safe partition with the finest possible granularity and then *grouping* partitions, larger partitions may be formed. Any one of these groupings may expose the optimal parallelization of the loop. Unfortunately, there exists an exponential number of possible groupings [2].

To limit the search space, statement order is fixed based on a topological sort of all the dependences for $\mathcal{L}$. Ambiguities are resolved in favor of placing parallel partitions adjacent to each other. The advantage of this ordering is that loop-carried anti-dependences may be broken, allowing parallelism to be exposed.

Grouping partitions via dynamic programming. A dynamic programming solution is used to compute the best grouping for the finest granularity ordered partitions. This algorithm is similar to techniques for calculating the shortest path between two points in a graph [31]. The algorithm is $O(N * M^3)$. N is the number of perfectly nested loops. M is the maximum number of partitions and is less than or equal to the number of statements in the loop. Both N and M are typically small numbers.

The dynamic programming solution appears in Figure 3. The algorithm begins by finding the finest partition for the inner loop l_n that satisfies its own dependences and the ordering constraints. On subsequent iterations, the initial partition is further constrained by including the dependences for the next outer loop. Since an inner loop may have more partitions than its enclosing loop, a map is constructed that correlates a statement's partition for the previous and current iteration; $map(j)$ returns the partition from l_{i+1} that corresponds to π_j in l_i.

For each loop level, *BestCost* calculates the best execution time of each possible grouping of partitions. The grouping algorithm first tests the finest partition and then each pair of adjacent partitions. Increasingly larger groupings of partitions are tested for a particular loop level. At each level, the minimal execution time for each grouping analyzed is stored. The minimal grouping time is taken from the grouping at this level, as well as that of the previous inner loops. This strategy allows inner loop distributions to be used within an outer loop distribution to minimize overall execution time. On completion, the best execution time for the grouping of the entire loop nest is determined.

Input:
$\mathcal{L} = \{l_1, \ldots, l_n\}$ perfect loop nest
$S = \{s_1, \ldots, s_p\}$ ordered body of $\mathcal{L}$
$IT = \{it_1, \ldots, it_n\}$ number of loop iterations
$time^{(i)}_{j,k} = BestCost\ (\{\pi_j, \ldots, \pi_k\}, \{l_i, \ldots, l_n\})$
Output:
$opt^{(i)}_{j,k} = \min_{j \le r \le k}(time^{(i)}_{j,r} + time^{(i)}_{r+1,k})$
best execution time for l_i
$D^{(i)}_{j,k}$ = grouping of partitions at l_i
with best execution time
Grouping via dynamic programming:
for $i = n, 1, -1$
partition into $\pi_1, \ldots, \pi_m$
for $\delta = 0, m-1$
for $j = 1, m-\delta$
$opt^{(i)}_{j,j+\delta} = min(time^{(i)}_{j,j+\delta}, it_{i+1} * time^{(i+1)}_{map(j),map(j+\delta)})$
if $(time^{(i)}_{j,j+\delta} < time^{(i+1)}_{map(j),map(j+\delta)})$ **then**
$D^{(i)}_{j,j+\delta} = \{\{\pi_j, \ldots, \pi_{j+\delta}\}\}$
else
$D^{(i)}_{j,j+\delta} = D^{(i)}_{map(j),map(j+\delta)}$
endif
for $k = 0, \delta - 1$
if $(opt^{(i)}_{j,j+\delta} > opt^{(i)}_{j,j+k} + opt^{(i)}_{j+k+1,j+\delta})$ **then**
$opt^{(i)}_{j,j+\delta} = opt^{(i)}_{j,j+k} + opt^{(i)}_{j+k+1,j+\delta}$
$D^{(i)}_{j,j+\delta} = D^{(i)}_{j,j+k} \cup D^{(i)}_{j+k+1,j+\delta}$
endif
endfor
endfor
endfor
endfor

Figure 3:

Each time the algorithm locates a grouping of partitions that improves execution time, a set D is constructed to describe how partitions are grouped together. For a loop l_i, $D^{(i)}_{1,m}$ provides the best grouping of partitions at loop l_i. Upon termination of the algorithm, $D^{(1)}_{1,m}$ indicates the final grouping with the minimal cost. Implicit in D is also a description of any additional transformations specified by *BestCost*.

Improvements. To leverage the dynamic programming solution, the distribution algorithm generates partitions based on a fixed statement order that satisfies all the dependences. A correct and less restrictive statement order uses only the dependences for the particular loop nest being distributed. In general, this ordering causes the map between solutions for adjacent loop partitions to be useless. It provides a single best solution for each nesting level of distribution instead of one overall best solution. In practice, experimentation will be needed to differentiate these strategies.

6 Experimental Validation

This section presents significant performance improvements due to interprocedural transformation on two scientific programs, spec77 and ocean, taken from the Perfect Benchmarks[16]. Spec77 contains 3278 non-comment lines and is a fluid dynamics weather simulation that uses Fast Fourier Transforms and rapid elliptic problem solvers. Ocean has 1902 non-comment lines and is a 2-D fluid dynamics ocean simulation that also uses Fast Fourier Transforms.

To locate opportunities for transformations, we browsed the dependences in the program using the ParaScope Editor [6, 25, 26]. Using other ParaScope tools, we determined which procedures in the program contained procedure calls. We examined the procedures containing calls, looking for interesting call structures. We located adjacent calls, loops adjacent to calls, and loops containing calls which could be optimized.

The rest of this section describes our experiences executing these programs on a 20-processor Sequent Symmetry S81. Since the optimizations used and the experimental methodology differed slightly for each program, they are described separately.

6.1 Optimizing spec77

In spec77, loops containing calls were common. Overall, transformations were applied to 19 such loops. Embedding and interchange were applied to 8 loops which contained calls to a single procedure. The remaining 11 loops, which contained multiple procedure calls, were optimized using extraction, fusion and interchange. These loops were found in procedures *del4*, *gloop* and *gwater*.

For the 19 transformed loops, performance was measured among three possibilities: (1) no parallelization of loops containing procedure calls, (2) parallelization using interprocedural information, and (3) interprocedural information and transformations. To obtain these versions, the steps illustrated in Figure 4 were performed.

The Original version contains directives to parallelize the loops in the leaf procedures that are invoked by the 19 loops of interest. The IPinfo version parallelizes the 19 loops containing calls. For the IPtrans version, we performed interprocedural transformation followed by outer loop parallelization. The parallel loops in each version were also blocked to allow multiple consecutive iterations to execute on the same processor without synchronization. The compiler default is to create a separate process for each iteration of a parallel loop.

	Time in optimized portion	Speedup
Processors = 7		
Original	81.9s	5.7
IPinfo	80.0s	5.8
IPtrans	80.6s	5.8
Processors = 19		
Original	45.8s	10.1
IPinfo	48.0s	9.7
IPtrans	36.4s	12.7

The results reported above are the best execution time in seconds for the optimized portions of each version. The speedups are compared against the execution time in the optimized portion of the program on a single processor, which was 463.7s. This accounted for more than 21 percent of the total sequential execution time.

With seven processors, the results are similar for all three versions, since each program version provided adequate parallelism and granularity for seven processors. On 19 processors, IPinfo was slower than the original program because the parallel outer loops had insufficient parallelism – only 7 to 12 iterations. The parallel inner loops of Original were better matched to the number of processors because they had at least 31 iterations. The interprocedural transformation version IPtrans demonstrated the best performance, a speedup of 12.7, because it combined the amount of parallelism in Original with increased granularity. The interprocedural transformations resulted in a 21 percent improvement in execution time over Original in the optimized portion.

Parallelizing just these 19 loops resulted in a speedup for the entire program of about 1.25 on 19 processors and 1.23 on 7 processors. Higher speedups might result from parallelizing the entire application.

6.2 Optimizing ocean

There were 31 places in the main routine of ocean where we extracted and fused interprocedurally adjacent loops. They were divided almost evenly between adjacent calls and loops adjacent to calls. In all 15 cases where a loop was adjacent to a call, the loop was 2-dimensional, while the loop in the called procedure was 1-dimensional. Prior to fusion, we *coalesced* the 2-dimensional loop into a 1-dimensional loop by linearizing the subscript expressions of its array references. The resulting fused loops consisted of between 2 and 4 parallel loops from the original program, thus increasing the granularity of parallelism.

To measure performance improvements due to interprocedural transformation, we performed steps similar to those in Figure 4. Directives forced the parallelization and blocking of the individual loops in the Original version, and the fused loops in IPtrans. The execution times were measured for the entire program and just the optimized portion. The optimized execution times are shown below.

	Processors = 19	
	Time in optimized portion	Speedup
Original	116.6s	5.5
IPtrans	79.3s	8.1

The speedups are relative to the time in the optimized portion of the sequential version of the program, which was 645.9 seconds. The optimized code accounted for about 5 percent of total program execution time. For the whole program, the parallelized versions achieve a speedup of about 1.06 over the sequential execution time.

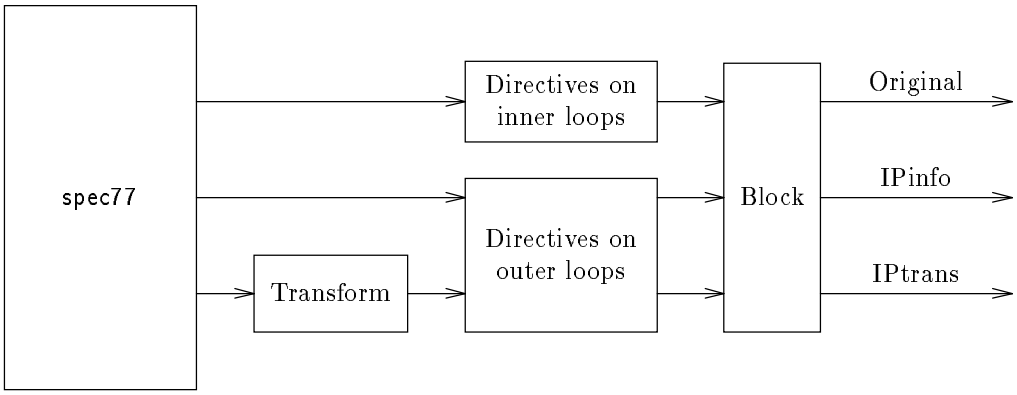

Figure 4: Stages of preparing program versions for experiment.

Note that IPtrans achieved a 32 percent improvement over Original in the optimized portion. This improvement resulted from increasing the granularity of parallel loops and reducing the amount of synchronization. It is also possible that fusion reduced the cost of memory accesses. Often the fused loops were iterating over the same elements of an array. These 31 groups of loops were not the only opportunities for interprocedural fusion; there were many other cases where fusion was safe, but the number of iterations were not identical. Using a more sophisticated fusion algorithm might result in even better execution time improvements.

7 Related Work

While the idea of interprocedural optimization is not new, previous work on interprocedural optimization for parallelization has limited its consideration to inline substitution [4, 13, 23] and interprocedural analysis of array side effects [5, 9, 12, 20, 29, 30, 35]. The various approaches to array side-effect analysis must make a tradeoff between precision and efficiency. Section analysis used here loses precision because it only represents a few array substructures, and it merges sections for all references to a variable into a single section. However, these properties make it efficient enough to be widely used by code generation. In addition, experiments with regular section analysis on the LINPACK library demonstrated a 33 percent reduction in parallelism-inhibiting dependences, allowing 31 loops containing calls to be parallelized [20]. Comparing these numbers against published results of more precise techniques, there was no benefit to be gained by the increased precision of the other techniques [29, 30, 35].

Sections inspired a similar but more detailed array summary analysis, data access descriptors, which stores access orders and expresses some additional shapes [5, 21, 22]. In fact, the slice annotation to sections could be obviated by using some of the techniques in Huelsbergen et. al. for determining exact array descriptors for use in dependence testing. However, slices are appealing due to our existing implementation and their simplicity.

8 Conclusions

This paper has described a compilation system; introduced two interprocedural transformations, loop embedding and loop extraction; and proposed a parallel code generation strategy. The usefulness of this approach has been illustrated on the Perfect Benchmark programs spec77 and ocean. Taken as a whole, the results indicate that providing freedom to the code generator becomes more important as the number of processors increase. Effectively utilizing more processors requires more parallelism in the code. This behavior was particularly observed in spec77, where the benefits of interprocedural transformation were increased with the number of processors.

Although it may be argued that scientific programs structured in a modular fashion are rare in practice, we believe that this is an artifact of the inability of previous compilers to perform interprocedural optimizations of the kind described here. Many scientific programmers would like to program in a more modular style, but cannot afford to pay the performance penalty. By providing compiler support to effectively optimize procedures containing calls, we encourage the use of modular programming, which, in turn, will make these transformations applicable on a wider range of programs.

Acknowledgments

We are grateful to Paul Havlak, Chau-Wen Tseng, Linda Torczon and Jerry Roth for their contributions to this work. Use of the Sequent Symmetry S81 was provided by the Center for Research on Parallel Computation under NSF Cooperative Agreement # CDA8619893.

References

[1] F. Allen and J. Cocke. A catalogue of optimizing transformations. In J. Rustin, editor, *Design and Optimization of Compilers*. Prentice-Hall, 1972.

[2] J. R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Proceedings of the Fourteenth Annual ACM Symposium on the Principles of Programming Languages*, Munich, Germany, January 1987.

[3] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Pro-*

gramming Languages and Systems, 9(4):491–542, October 1987.

[4] R. Allen and S. Johnson. Compiling C for vectorization, parallelization, and inline expansion. In *Proceedings of the SIGPLAN '90 Conference on Program Language Design and Implementation*, Atlanta, GA, June 1990.

[5] V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *Proceedings of the SIGPLAN '89 Conference on Program Language Design and Implementation*, Portland, OR, June 1989.

[6] V. Balasundaram, K. Kennedy, U. Kremer, K. S. McKinley, and J. Subhlok. The ParaScope Editor: An interactive parallel programming tool. In *Proceedings of Supercomputing '89*, Reno, NV, November 1989.

[7] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston, MA, 1988.

[8] P. Briggs, K. Cooper, M. W. Hall, and L. Torczon. Goal-directed interprocedural optimization. Technical Report TR90-147, Dept. of Computer Science, Rice University, December 1990.

[9] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.

[10] D. Callahan, J. Cocke, and K. Kennedy. Estimating interlock and improving balance for pipelined machines. *Journal of Parallel and Distributed Computing*, 5(4):334–358, August 1988.

[11] D. Callahan, K. Cooper, R. Hood, K. Kennedy, and L. Torczon. ParaScope: A parallel programming environment. *The International Journal of Supercomputer Applications*, 2(4):84–99, Winter 1988.

[12] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. In *Proceedings of the First International Conference on Supercomputing.* Springer-Verlag, Athens, Greece, June 1987.

[13] K. Cooper, M. W. Hall, and L. Torczon. An experiment with inline substitution. *Software—Practice and Experience*, 21(6):581–601, June 1991.

[14] K. Cooper, K. Kennedy, and L. Torczon. The impact of interprocedural analysis and optimization in the $\mathbb{R}^n$ programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):491–523, October 1986.

[15] K. Cooper, K. Kennedy, and L. Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.

[16] G. Cybenko, L. Kipp, L. Pointer, and D. Kuck. Supercomputer performance evaluation and the Perfect benchmarks. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[17] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[18] G. Goff, K. Kennedy, and C. Tseng. Practical dependence testing. In *Proceedings of the SIGPLAN '91 Conference on Program Language Design and Implementation*, Toronto, Canada, June 1991.

[19] M. W. Hall. *Managing Interprocedural Optimization.* PhD thesis, Dept. of Computer Science, Rice University, April 1991.

[20] P. Havlak and K. Kennedy. Experience with interprocedural analysis of array side effects. In *Proceedings of Supercomputing '90*, New York, NY, November 1990.

[21] L. Huelsbergen, D. Hahn, and J. Larus. Exact dependence analysis using data access descriptors. In *Proceedings of the 1990 International Conference on Parallel Processing*, St. Charles, IL, August 1990.

[22] L. Huelsbergen, D. Hahn, and J. Larus. Exact dependence analysis using data access descriptors. Technical Report 945, Dept. of Computer Science, University of Wisconsin at Madison, July 1990.

[23] C. A. Huson. An inline subroutine expander for Parafrase. Master's thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1982.

[24] K. Kennedy and K. S. McKinley. Loop distribution with arbitrary control flow. In *Proceedings of Supercomputing '90*, New York, NY, November 1990.

[25] K. Kennedy, K. S. McKinley, and C. Tseng. Analysis and transformation in the ParaScope Editor. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[26] K. Kennedy, K. S. McKinley, and C. Tseng. Interactive parallel programming using the ParaScope Editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):329–341, July 1991.

[27] D. Kuck. *The Structure of Computers and Computations, Volume 1.* John Wiley and Sons, New York, NY, 1978.

[28] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. J. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth Annual ACM Symposium on the Principles of Programming Languages*, Williamsburg, VA, January 1981.

[29] Z. Li and P. Yew. Efficient interprocedural analysis for program restructuring for parallel programs. In *Proceedings of the ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems (PPEALS)*, New Haven, CT, July 1988.

[30] Z. Li and P. Yew. Interprocedural analysis and program restructuring for parallel programs. Technical Report 720, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, January 1988.

[31] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IRE Transactions on Electronic Computers*, 9(1):39–47, 1960.

[32] Y. Muraoka. *Parallelism Exposure and Exploitation in Programs.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, February 1971. Report No. 71-424.

[33] C. Polychronopoulos. *On Program Restructuring, Scheduling, and Communication for Parallel Processor Systems.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, August 1986.

[34] V. Sarkar. *Partition and Scheduling Parallel Programs for Multiprocessors.* The MIT Press, Cambridge, MA, 1989.

[35] R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of CALL statements. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.

[36] M. J. Wolfe. Loop skewing: The wavefront method revisited. *International Journal of Parallel Programming*, 15(4):279–293, August 1986.

[37] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge, MA, 1989.