RICE UNIVERSITY

Automatic and Interactive Parallelization

by

Kathryn S. McKinley

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Ken Kennedy
Noah Harding Professor, chair
Computer Science

Keith D. Cooper, Associate Professor
Computer Science

Don H. Johnson, Professor
Electrical and Computer Engineering

Danny C. Sorensen, Professor
Mathematical Sciences

Houston, Texas

March, 1994

Automatic and Interactive Parallelization

Kathryn S. McKinley

Abstract

The goal of this dissertation is to give programmers the ability to achieve high performance by focusing on developing parallel algorithms, rather than on architecture-specific details. The advantages of this approach also include program portability and legibility. To achieve high performance, we provide automatic compilation techniques that tailor parallel algorithms to shared-memory multiprocessors with local caches and a common bus. In particular, the compiler maps complete applications onto the specifics of a machine, exploiting both parallelism and memory.

To optimize complete applications, we develop novel, general algorithms to transform loops that contain arbitrary conditional control flow. In addition, we provide new interprocedural transformations which enable optimization across procedure boundaries. These techniques provide the basis for a robust automatic parallelizing algorithm that is applicable to complete programs.

The algorithm for automatic parallel code generation takes into consideration the interaction of parallelism and data locality, as well as the overhead of parallelism. The algorithm is based on a simple cost model that accurately predicts cache line reuse from multiple accesses to the same memory location and from consecutive accesses. The optimizer uses this model to improve data locality. It also uses the model to discover and introduce effective parallelism that complements the benefits of data locality. The optimizer further improves the effectiveness of parallelism by seeking to increase its granularity. Parallelism is introduced only when granularity is sufficient to overcome its associated costs.

The algorithm for parallel code generation is shown to be efficient and several of its component algorithms are proven optimal. The efficacy of the optimizer is illustrated with experimental results. In most cases, it is very effective and either achieves or improves the performance of hand-crafted parallel programs. When performance is not satisfactory, we provide an interactive parallel programming tool which combines compiler analysis and algorithms with human expertise.

Acknowledgments

Ken Kennedy provided me with the three most important elements of support in graduate school: intellectual, political and financial. In addition, Ken, Keith Cooper and Linda Torczon fostered a research atmosphere and working environment whose benefits are untold. I would also like to recognize the other members of my committee, Keith Cooper, Don Johnson and Danny Sorensen. Keith has been an endless source of encouragement and wisdom throughout my graduate career. Don Johnson gave me my first taste of research and hooked me for life.

I am fortunate that many of my fellow graduate students and friends supported my research intellectually, emotionally and with implementations. I would especially like to thank Chau-Wen Tseng, Marina Kalem, Mary Hall, Paul Havlak, Nat McIntosh, Preston Briggs, Ben Chase and the entire compiler group.

I am extremely gratefully to my entire family. As always, my parents were a constant source of love and encouragement. To my husband Scotty Strahan, I hope I am the rock for you that you have been for me.

A little Madness in the Spring
Is wholesome even for the King. Emily Dickinson (1830-1886).

Contents

Illustrations

Chapter 1

Introduction

Many program transformations that introduce parallelism into sequential, scientific Fortran programs have proven effective in improving performance on vector and shared-memory multiprocessor hardware. For advanced parallel architectures, obtaining the best performance often requires the program to be modified for the particular features of the underlying architecture. Currently, users must modify their programs for each architecture of interest to achieve high performance. Not only are programmers required to understand architecture specific details, their programs are usually not portable once they have been modified in this fashion. To address these problems, this dissertation seeks to determine the following:

> *Does there exists a machine-independent parallel programming style from which compilers can produce parallel programs with acceptable or excellent performance on shared-memory multiprocessors with local caches and a common bus?*

Clearly, we are not attempting to solve the "dusty deck" problem, where a program developed using a sequential algorithm is automatically transformed to a parallel one. This problem is inherently more difficult because programs may need significant technical expertise or algorithmic restructuring for good parallel performance. In fact, this problem has not been solved even for uniprocessor vector machines [KKLW80b, CDL88].

A lesson to be learned from vectorization is that programmers rewrote their programs in a portable, vectorizable style based on feedback from vectorizing compilers [CKK89, Wol89c]. Compilers were then able to take these programs and generate machine-dependent vector code with excellent results. We are testing this same thesis for the harder problem of shared-memory parallel machines.

Vectorization achieves high performance by simply utilizing parallelism on a single statement for a single loop level. On any architecture where parallelism exacts a higher cost, larger regions of parallelism, *i.e. higher granularity*, must be discovered and exploited to achieve high performance. Because successes in this arena were few, we choose to explore parallel code generation for shared-memory multiprocessors with local caches and a common bus. We believe the solution to this problem to be a first step in compiling for more advanced parallel architectures.

We advocate that one machine independent program version be developed in a sequential language, such as Fortran 77, the most widely used programming language in the scientific community. Our compiler would then apply ambitious algorithms to customize the program for a shared-memory multiprocessor.

1.1 Automatic parallelization

Previous automatic parallel code generation algorithms for shared-memory multiprocessors are, for the most part, *ad hoc* and have not yet established an acceptable level of success. Although, many transformations and combinations of transformations have been shown to parallelize interesting example loops, an effective overall parallelization strategy for complete applications has not been forthcoming [ABC$^+$88, ACK87, KKLW80a, Wol89a, WL90]. The automatic parallelization problem is very difficult for a variety of reasons.

One important reason is that the theoretical statement of seeking all possible parallelism does not work well in practice. In practice, parallelism incurs overhead. If this overhead is not taken into account, parallelization can degrade performance rather than enhance it. Similarly, parallelism introduced without regard to its effect on the performance of the memory subsystem can degrade performance. Another reason parallelization is difficult to discover in complete applications is that it requires precise array analysis in the presence of procedure calls. Until recently, this analysis was not available [CK87b, HK90, HK91].

We have developed a new interprocedural approach for automatic parallel code generation for complete applications. Two important components of this algorithm are generalized and interprocedural transformations that attack the problems found real programs.

1. *Generalized transformations for loops containing conditional control flow.* Much previous work cannot apply parallelizing transformations when loops contain conditional control flow. In this thesis, a broad selection of loop transformations is extended to deal with conditional branches using the control dependence representation. In particular, a new algorithm for performing loop distribution is shown to be optimal for a legal partitioning of the statements into new loops.

2. *Interprocedural transformations.* We introduce two new interprocedural transformations, *loop embedding* and *loop extraction* that expose loop nests to other optimizations without incurring costs associated with procedure inlining. We present a strategy for determining the benefits and safety of these two transformations when combined with other loop-based optimizations. The compound transformations are judiciously applied when performance is expected to improve. The recompilation system and analysis needed to perform and test these optimizations is shown to be efficient.

These algorithms enable automatic parallelization of complete applications.

Three important factors in optimizing for parallel architectures are granularity, parallelism and data locality. Parallelism is usually most effective when it achieves the highest possible granularity, the amount of work per parallel task. The granularity of parallelism must also be sufficient to overcome the overhead of parallelism, such as processor synchronization costs. We only perform loops in parallel when performance estimation determines there is enough granularity to improve execution time.

To address data locality, we present a simple cost model for determining cache line reuse. It computes reuse due to accesses to consecutive memory locations on a particular cache line and reuse due to multiple accesses to the same memory location on a cache line. The cost model is used to order loops in a nest to improve data locality and to discover and exploit parallelism. This optimization strategy produces data locality at the innermost loops and parallelism at the outermost loop. Each is placed where it is most likely to be effective. Experimental results validate this approach. They indicate that the cost model is accurate and effective for driving optimization, even for scalar machines.

This strategy provides the core of the optimizer. Algorithms for applying additional loop transformations are also described. In particular, a new, unified algorithm for performing loop fusion and distribution is presented which achieves maximal granularity under certain constraints. Several component algorithms are shown to be optimal. The optimization algorithms are based on theoretical and practical considerations. All of the algorithms are incorporated into a cohesive interprocedural parallel code generation algorithm.

1.2 Interactive parallelization

Unfortunately, automatic parallelization is unlikely to yield excellent parallel performance in every case for a variety of reasons. For example, the algorithm may be unsuitable for parallel execution. One difficulty which often arises is that important constants and symbolics values are unknown at compile time. Therefore, in addition to improved automatic parallelization via advanced compiler techniques, we combine compiler strategies and human insight in an interactive parallel programming tool.

Our tool, the ParaScope Editor is intended to provide all the analysis and optimization capabilities of the parallelizer in an intelligent editor. It provides a large collection of parallelism enhancing transformations that have proven effective, such as loop interchange, loop fusion and strip mining. It contains a user-assertion facility for communication between the user and the compiler, as well as an advanced text and structure editor for Fortran with functions such as searching and view filtering. In order to make the ParaScope Editor (Ped) truly interactive, updates after user changes such as edits, transformations and assertions must be quick and precise. We describe fast incremental algorithms for precise updates after user or compiler changes. They are implemented in Ped and have proven themselves efficient in practice [HHK+93].

1.3 Overview

In Chapter 2 we begin by describing the analysis required to perform effective program parallelization. We then serve two purposes by discussing the ParaScope Editor in Chapter 3. The first is a general description of interactive parallel programming and the supporting implementation. However, we also introduce the loop-based transformations which form a basis for parallel code generation. We detail several of the transformation algorithms and present new incremental update algorithms for them. These algorithms serve as an introduction to the analysis and representations necessary to support automatic as well as interactive parallelization.

The next two chapters are devoted to new algorithms that make the parallelization of complete applications viable throughout the rest of the dissertation. Chapters 6 and 7 then develop an integrated, automatic parallel code generation strategy. This optimizer is tested experimentally to determine if it provides support for machine-independent parallel programming.

The experiment compares a good hand-coded parallel program to one derived by hand-simulating our automatic algorithm on a nearby sequential version. Therefore, parallelism is known to exist and we measure the ability of our automatic techniques to uncover this parallelism. The results of our experiment indicate that given a few assertions, the automatically generated versions usually perform as well or better than hand-coded versions. These results do not completely prove the thesis statement, but provide very promising support for it.

Chapter 2

Technical Background

For the most part, this dissertation focuses on on exploiting existing analysis to perform effective optimizing transformations. To understand these optimizations requires knowledge of the tenents on which they are based. We therefore begin with an overview of the analysis required for program parallelization and transformation.

2.1 Dependence Analysis

Dependences describe a partial order between statements that must be maintained to preserve the meaning of a program with sequential semantics. A dependence between statement S_1 and S_2, denoted $S_1 \delta S_2$, indicates that S_1, the *source*, must be executed before S_2, the *sink*. There are two types of dependence: data dependence and control dependence.

Data dependence

A *data dependence*, $S_1 \delta S_2$, indicates that S_1 and S_2 read or write a common memory location in a way that requires their execution order to be preserved [Ber66]. There are four types of data dependence [Kuc78]:

True (flow) dependence
occurs when S_1 writes a memory location that S_2 later reads.

Anti dependence
occurs when S_1 reads a memory location that S_2 later writes.

Output dependence
occurs when S_1 writes a memory location that S_2 later writes.

Input dependence
occurs when S_1 reads a memory location that S_2 later reads.[1]

[1]Input dependences do not restrict statement order.

Control dependence

Intuitively, a *control dependence*, $S_1 \delta_c S_2$, indicates that the execution of S_1 directly determines whether S_2 will be executed. The control flow graph G_f represents the flow of execution in the program. The following formal definitions of control dependence and the postdominance relation computed on G_f are taken from the literature [FOW87, CFS90].

> **Definition 2.1** x is *postdominated* by y in G_f if every path from x to the exit node of G_f contains y.

> **Definition 2.2** Given two statements x, $y \in G_f$, y is *control dependent* on x if and only if:
>
> 1. $\exists$ a non-null path p, $x \rightarrow y$, such that y postdominates every node between x and y on p, and
> 2. y does not postdominate x.

Based on these definitions, a control dependence graph G_{cd} can be built with the control dependence edges $(x,\ y)_l$ where l is the label of the first edge on path $x \rightarrow y$. Additionally, if G_f is structured, rooted and acyclic, the resulting G_{cd} is a tree, where structured has its usual meaning as originally formulated by Böhm and Jacopini [BJ66]. If G_f is unstructured, rooted and acyclic, the resulting G_{cd} is a DAG [CFS90].

Loop-carried and loop-independent dependence

Because scientific Fortran programs spend most of their time executing loops [Knu71], this thesis focuses on executing loops in parallel. Dependence analysis determines which loops in the program may be run safely in parallel. A dependence between iterations of a loop is called *loop-carried* and prevents the iterations of a loop from being executed in parallel [All83, AK87]. Consider the following loop:

```
        DO I = 2, N
S1          A(I)    = ...
S2          ...     = A(I)
S3          ...     = A(I-1)
        ENDDO
```

The true dependence $S_1 \delta S_2$ is called *loop-independent* because it exists regardless of the surrounding loops. Loop-independent dependences, whether data or control, occur within a single iteration of a loop and do not inhibit a loop from running in parallel. For example, if $S_1 \delta S_2$ were the only dependence in the loop, the iterations of this loop could be run in parallel, because statements executed on each iteration only affect other statements in the same iteration and not in any other iterations. However, loop-independent dependences do affect statement order within a loop iteration. Interchanging statements S_1 and S_2 violates the loop-independent dependence and changes the meaning of the program.

By comparison, the true dependence $S_1 \delta S_3$ is *loop-carried* because the source and sink of the dependence occur on different iterations of the loop; S_3 reads the memory location that was written by S_1 on the previous iteration. Loop-carried dependences inhibit loop iterations from executing in parallel without explicit synchronization. When there are nested loops, the *level* of any carried dependence is the outermost loop on which it first arises [All83, AK87].

Dependence testing

Determining the existence of data dependence between array references is more difficult than for scalars, because the subscript expressions must be considered. The process of differentiating between two subscripted references in a loop nest is called *dependence testing* [Ban88, Wol89b, GKT91]. To illustrate, consider the problem of determining whether or not there exists a dependence from statement S_1 to S_2 in the following loop nest:

```
     DO i1 = L1, U1
         DO i2 = L2, U2
             ...
             DO in = Ln, Un
S1                 A(f1(i1, ..., in), ..., fm(i1, ..., in)) = ...
S2                 ... = A(g1(i1, ..., in), ..., gm(i1, ..., in))
             ENDDO
             ...
         ENDDO
     ENDDO
```

Let α and β be vectors of n integer indices within the ranges of the upper and lower bounds of the n loops. There is a dependence from S_1 to S_2 if and only if there exist α and β such that α is lexicographically less than or equal to β and the following system of *dependence equations* is satisfied:

$$f_k(\alpha) \;=\; g_k(\beta) \quad \forall k,\ 1 \leq k \leq m$$

Distance and direction vectors

Distance and direction vectors may be used to characterize data dependences by their access pattern between loop iterations. If there exists a data dependence for $\alpha = (\alpha_1, \ldots, \alpha_n)$ and $\beta = (\beta_1, \ldots, \beta_n)$, then the *distance vector* $\mathbf{D} = (D_1, \ldots, D_n)$ is defined as $\beta - \alpha$. The *direction vector* $\mathbf{d} = (d_1, \ldots, d_n)$ of the dependence is defined by the equation:

$$d_i = \begin{cases} < & \text{if } \alpha_i < \beta_i \\ = & \text{if } \alpha_i = \beta_i \\ > & \text{if } \alpha_i > \beta_i \end{cases}$$

Dependence distances and directions are represented as a vector whose elements, displayed left to right, represent the dependence from the outermost to the innermost loop in the nest. By definition all distance and direction vectors are lexicographically positive. We use

$$\vec{\delta} = (\delta_1, \ldots, \delta_n)$$

to represent a distance or direction vector, where δ_i is the dependence distance or direction for the loop at level i. For example, consider the following loop nest:

```
DO I = 1, N
     DO J = 1, M
          DO K = 1, l
               A(I+1, J, K-1) = A(I, J, K) + C
          ENDDO
     ENDDO
ENDDO
```

The distance and direction vectors for the true dependence between the definition and use of array A are $(1, 0, -1)$ and $(<, =, >)$, respectively. Since several different values of α and β may satisfy the dependence equations, a set of distance and direction vectors may be needed to completely describe the dependences arising between a pair of array references.

Distance vectors, first used by Kuck and Muraoka [KMC72, Mur71], specify the number of loop iterations between two accesses to the same memory location. Direction vectors, introduced by Wolfe [Wol82], summarize distance vectors and are therefore less precise. However, there are situations where direction vectors may be computed, but distance vectors cannot be.

Both may be used to calculate loop-carried dependences. Additionally, direction vectors are sufficient to determine the safety and profitability of loop interchange [AK87, Wol82]. Distance vectors are often required by other transformations that exploit parallelism [Ban90b, KMT91a, Lam74, WL90, Wol86] and improve data locality [CCK90, KMT91a, GJG87]. Data dependence also characterizes reuse of individual memory locations [CCK90].

2.2 Interprocedural dependence analysis

The presence of procedure calls complicates the process of analyzing dependences. Without interprocedural analysis worst case assumptions must be made in the presence of procedure calls. Conventional interprocedural analysis discovers constants, aliasing, flow-insensitive side effects such as REF and MOD, and flow-sensitive side effects such as USE and KILL [CCKT86, CKT86a]. However, parallelization is limited because arrays are treated as monolithic objects, making it impossible to determine whether two references to an array actually access the same memory location.

Array sections

To provide more precise analysis, array accesses can be summarized in terms of *regular sections* or *data access descriptors* that describe subsections of arrays such as rows, columns and rectangles [BK89, CK87b, HK91]. Local symbolic analysis and interprocedural constants are required to build accurate sections. Once constructed, sections may be quickly intersected during interprocedural analysis and dependence testing to determine whether dependences exist. This analysis is described in more detail in Section 5.2.2.

2.3 Augmented call graph

The program representation for our work on whole program optimization requires an *augmented call graph* to describe the calling relationship among procedures and specify loop nests. For this purpose, the program's call graph, which contains the usual *procedure nodes* and *call edges*, is augmented to include *loop nodes* and *nesting edges*. The loop nodes contain loop header information. 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. Each loop is also marked as sequential or parallel. A loop with no loop-carried dependences (*i.e* all the direction vectors contain "=" for the loop) is parallel and all others sequential.

Chapter 3

Interactive Parallel Programming

The ParaScope Editor is a new kind of exploratory parallel programming tool for developing scientific Fortran programs. It is able to compensate in many cases for the deficiencies of automatic parallelizers, by bringing user expertise and compiler technology to bear on program parallelization. It assists the knowledgeable user by displaying the results of sophisticated program analyses and by providing a set of powerful interactive transformations. After an edit or parallelism-enhancing transformation, the ParaScope Editor incrementally updates both the analyses and source quickly. These fast updates are useful in both batch and automatic systems. This chapter focuses on these abilities and introduces the transformations and the analysis they require that are used throughout the thesis.

3.1 Introduction

The ParaScope Editor helps users interactively transform a sequential Fortran 77 program into a parallel program with explicit parallel constructs, such as those in PCF Fortran [Lea90]. In a language like PCF Fortran, the principal mechanism for the introduction of parallelism is the *parallel loop*, which specifies that its iterations may be run in parallel according to any schedule. The fundamental problem introduced by such languages is the possibility of nondeterministic execution. For example, consider converting the following sequential loop into a parallel loop.

```
DO I = 1, 100
     A(INDEX(I)) = A(INDEX(I)) + 1
ENDDO
```

Dependence analysis conservatively assumes that INDEX(I) for a particular iteration may equal INDEX(I) for a later iteration. Therefore, there may be a loop-carried dependence on A and an automatic parallelizer would not execute this loop in parallel. Unfortunately, a parallelizer is often forced to make conservative assumptions about whether dependences exist. These assumptions may arise because of complex subscripts (as above) or the use of unknown symbolics. As a result, automatic systems miss loops that could be parallelized. For example, it may be that INDEX(I) is a permutation, allowing the loop to be safely performed in parallel. This weakness has led previous researchers to conclude that automatic systems, by themselves, are not powerful enough to find all of the parallelism in a program.

However, the analysis performed by automatic systems can be extremely useful to the programmer during the parallelization process. The ParaScope Editor (PED) is based upon this observation. It is designed to support an interactive parallelization process in which the user examines a particular loop and its dependences. To safely parallelize a loop, the user must either determine that each dependence shown is not valid (because of some overly conservative assumption made by the system), or transform the loop to satisfy valid dependences. After each transformation, PED reconstructs the dependence graph so that the user may determine the level of success achieved and apply additional transformations if desired.

A tool with this much functionality is bound to be complex. PED incorporates a complete source editor and supports dependence analysis, dependence display, and a large variety of program transformations to enhance parallelism. We describe in detail elsewhere the usage, user interface and motivation of the ParaScope Editor [BKK+89, FKMW90, KMT91b]. We also cover elsewhere the types of analyses and representations needed to support this tool and automatic parallelization (see Section 2) [KMT91a]. In this chapter, we focus on efficient algorithms for incremental updates after a transformation or edit. All of these algorithms are implemented in PED.

We begin with an overview of the existing work model and a description of the transformation process. These descriptions include the mechanisms for communication between the user and PED, and an example PED session. The incremental algorithms for determining *safety* and *profitability*, and for performing the *update* of dependence information and source for four important transformations are also detailed. The transformations are loop interchange, loop skewing, loop distribution, and unroll and jam. We discuss related work and conclude.

3.2 Work Model

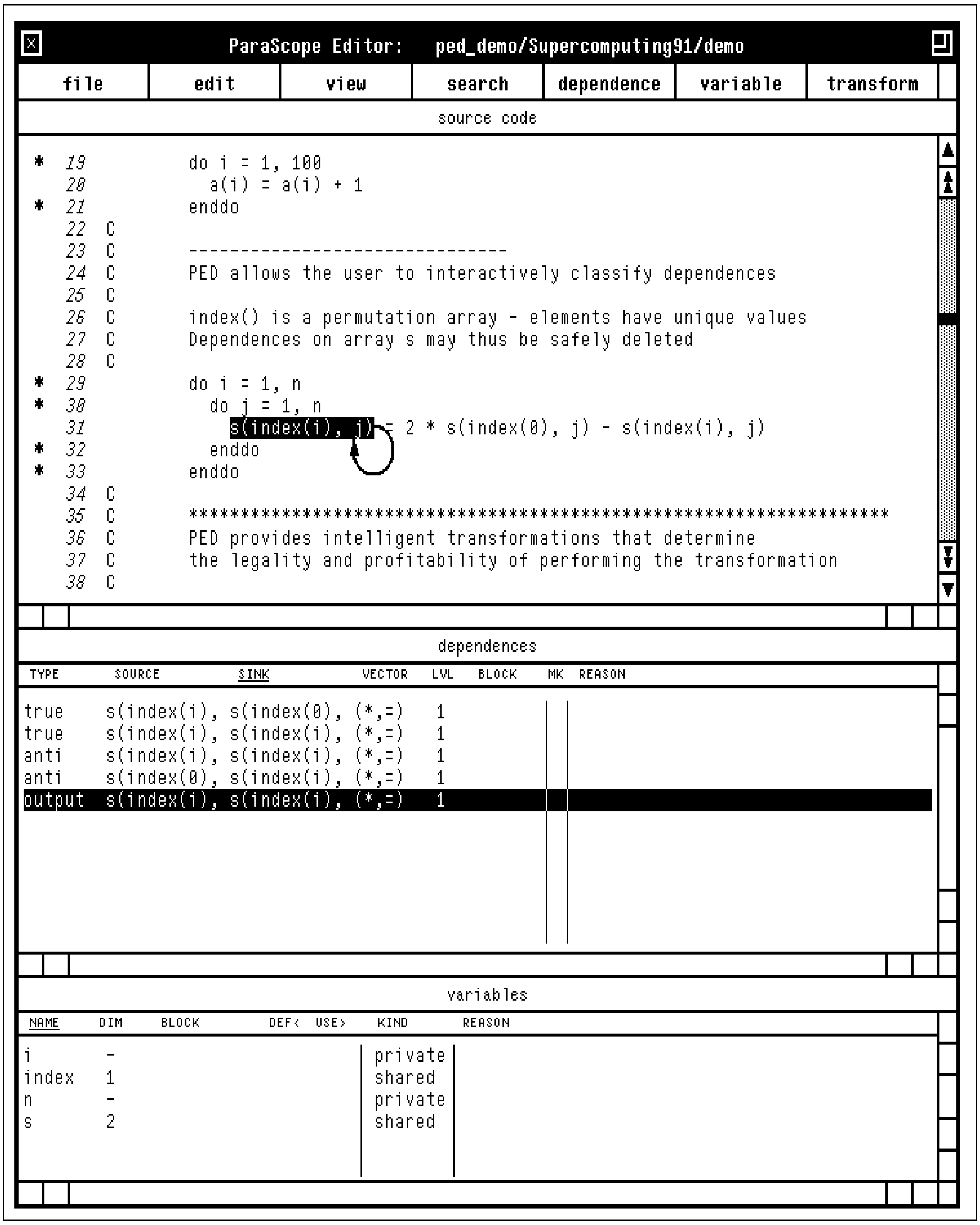
This thesis exploits *loop-level* parallelism, which comprises most of the usable parallelism in scientific codes when synchronization costs are considered [CSY90]. In the work model best supported by PED, the user first selects a loop for parallelization. PED then displays all of its loop-carried data dependences. Other dependences, such as control dependences, may also be displayed at the option of the user. The user may sort, filter or mark the dependences. This mechanism allows users to mark as *rejected* those dependences that are due to overly conservative dependence analysis, so that the transformations will ignore them. If dependence testing is exact and proves a dependence to exist, the dependence is pre-marked as *proven*. Otherwise, the dependence is pre-marked as *pending*, signifying to the user that it may be the result of overly-conservative analysis. Additionally, the user may mark pending dependences as *accepted*, indicating that the dependence does in fact occur. A similar facility is provided for variable classification [FKMW90]. These provide users with a powerful mechanism for experimenting with different parallelization strategies.

PED's user interface is shown in Figure 3.1. The figure shows a black and white screen dump of a color PED session. The *program pane* in the top half of the window displays a loop from a parallel direct search program produced by Virginia Torczon. The outer loop on line 29 is selected by clicking the mouse on the loop icon, the '*' in the leftmost column. The selection causes the header and all the enclosed statements to be a different color than the other text (in this black and white picture, it is not detectable). Buttons across the top of the editing pane invoke various PED features, such as transformations, program analysis, view filtering and editing.

In PED color is used to convey points of interest, focus or special meaning. The *dependence pane* is in the middle pane of the window and shows dependences carried by the selected loop. The output dependence on S(INDEX(I),J) is selected, which causes it to be highlighted in the dependence pane. The dependence is also reflected in the text pane by an arrow from the highlighted source reference to the highlighted sink reference. In this example, the dependence has its source and sink at the same reference, so only one reference is highlighted. If the end points of the dependence span the width of the screen, one end point is brought into view. To view the other end point the user need only select it in the dependence pane, and then PED will bring it into view in the text pane. The labels across the top of the dependence pane may be selected to sort by that characteristic. They may also be used in filter and marking queries on dependences.

The variable display at the bottom of the PED window presents each variable that participates in the loop. It also presents the classification of the variable if the loop were run in parallel. The user interface for all three of these displays is unified, requiring the user to learn only one simple paradigm.

Figure 3.1: **PED User Interface**



3.3 Transformations

PED provides a variety of interactive, structured transformations that enhance or expose parallelism in programs. If the user has made assertions about dependences and variables, the transformations take these into account. These transformations are applied according to a *power steering* paradigm: the user specifies the transformation to be made, and the system provides advice and carries out the mechanical details. The user is therefore relieved of the responsibility of making tedious and error prone program changes.

PED evaluates each transformation invoked according to three criteria: applicability, safety, and profitability. A transformation is *applicable* if it can be mechanically performed. For example, loop interchange is inapplicable for a single loop. A transformation is *safe* if it preserves the meaning of the original sequential program. Some transformations are always safe, others require a specific dependence pattern. Finally, PED classifies a transformation as *profitable* if it can determine that the transformation directly or indirectly improves the parallelism of the resulting program.

To perform a transformation, the user makes a program selection and invokes the desired transformation. If the transformation is inapplicable, PED responds with a diagnostic message. If the transformation is safe, PED advises the user as to its profitability. For parameterized transformations, PED may also suggest a parameter value. The user may then apply the transformation. For example, see loop skewing and unroll and jam in Sections 3.4.2 and 3.4.4.

If the transformation is unsafe or unprofitable, PED responds with a warning explaining the cause. In these cases, the user may decide to override the system advice and apply the transformation anyway. For example, if a user decides to parallelize a loop with loop-carried dependences, PED will warn the user of the dependences but allow the loop to be made parallel. This override ability is extremely important in an interactive tool, since it allows the user to apply knowledge unavailable to the tool. The program's abstract syntax tree (AST) and dependence information are automatically updated after each transformation to reflect the transformed source.

PED supports a large set of transformations that have proven useful for introducing, discovering, and exploiting parallelism. PED also supports transformations for improving data locality. Each transformation is briefly introduced below. Many are found in the literature [AC72, AK87, CCK90, KM90, KMT91b, KKLW84, Lov77, Wol86]. In PED, their novel aspect is the analysis of their applicability, safety, profitability and the incremental updates of source and dependence information. We classify the transformations implemented in PED as follows.

Reordering Transformations

Loop Distribution	Loop Interchange
Loop Skewing	Loop Reversal
Loop Fusion	Statement Interchange
Unroll and Jam	

Dependence Breaking Transformations

Privatization	Scalar Expansion
Array Renaming	Loop Peeling
Loop Splitting	Alignment

Memory Hierarchy Transformations

Strip Mining	Scalar Replacement
Loop Unrolling	

Miscellaneous Transformations

Sequential $\leftrightarrow$ Parallel	Loop Bounds Adjusting
Statement Addition	Statement Deletion

3.3.1 Reordering transformations

Reordering transformations change the order in which statements are executed, either within or across loop iterations. They are safe if all the dependences in the original program are preserved. Reordering transformations are used to expose or enhance loop-level parallelism. They are often performed in concert with other transformations to structure computations in a way that allows useful parallelism to be introduced. These may also be used to optimize data locality.

- **Loop distribution** partitions independent statements inside a loop into multiple loops with identical headers. It is used to separate statements that may be parallelized from those that must be executed sequentially [KM90, KMT91a, Kuc78]. The partitioning of the statements is targeted to vector or parallel hardware as specified by the user.

- **Loop interchange** interchanges the headers of two perfectly nested loops, changing the order in which the iteration space is traversed. When loop interchange is safe, it can be used to adjust the granularity of parallel loops [AK87, KMT91a, Wol89b].

- **Loop skewing** adjusts the iteration space of two perfectly nested loops by shifting the work per iteration in order to expose parallelism. When possible, PED computes and suggests the optimal skew degree. Loop skewing may be used with loop interchange in PED to expose wavefront parallelism [KMT91a, Wol86].
- **Loop reversal** reverses the order of execution of loop iterations.
- **Loop fusion** can increase the granularity of parallel regions and promote reuse by fusing two contiguous loops when dependences are not violated [AC72, KKP+81].
- **Statement interchange** interchanges two adjacent independent statements.
- **Unroll and jam** increases the potential candidates for *scalar replacement* and pipelining by unrolling the body of an outer loop in a loop nest and fusing the resulting inner loops [AC72, CCK90, CCK88, KMT91a].

3.3.2 Dependence breaking transformations

Dependence breaking transformations are used to satisfy specific dependences that inhibit parallelism. They may introduce new storage to eliminate storage-related anti or output dependences, or convert loop-carried dependences to loop-independent dependences, often enabling the safe application of other transformations. If all the dependences carried on a loop are eliminated, the loop may then be run in parallel.

- **Privatization** makes an array or scalar variable local to a parallel loop, eliminating dependences on the variable between loop iterations.
- **Scalar expansion** transforms a scalar variable into a one-dimensional array. It breaks output and anti dependences which may be inhibiting parallelism [KKLW80a].
- **Array renaming**, also known as node splitting [KKLW80a], is used to break anti dependences by copying the source of an anti dependence into a newly introduced temporary array and renaming the sink to the new array [AK87]. Loop distribution may then be used to separate the copying statement into a separate loop, allowing both loops to be parallelized.

- **Loop peeling** peels off the first or last k iterations of a loop as specified by the user. It is useful for breaking dependences which arise on the first or last k iterations of the loop [AC72].

- **Loop splitting**, or index set splitting, separates the iteration space of one loop into two loops, where the user specifies at which iteration to split. For example, if DO I = 1, 100 is split at 50, the following two loops result: DO I = 1, 50 and DO I = 51, 100. Loop splitting is useful in breaking *crossing dependences*, dependences that cross a specific iteration [AK87].

- **Alignment** moves instances of statements from one iteration to another to break loop-carried dependences [Cal87].

3.3.3 Memory hierarchy transformations

Memory optimizing transformations adjust a loop's balance between computations and memory accesses to make better use of the memory hierarchy and functional pipelines. These transformations have proven to be extremely effective for both scalar and parallel machines.

- **Strip mining** takes a loop with step size of 1, and changes the step size to a new user specified step size greater than 1. A new inner loop is inserted which iterates over the new step size. If the minimum distance of the dependences in the loop is less than the step size, the resultant inner loop may be parallelized. Used alone the order of the iterations is unchanged, but used in concert with loop interchange the iteration space may be *tiled* [Wol89a] to utilize memory bandwidth and cache more effectively [CK89].

- **Scalar replacement** takes array references with consistent dependences and replaces them with scalar temporaries that may be allocated into registers [CCK90]. It improves the performance of the program by reducing the number of memory accesses required.

- **Loop unrolling** decreases loop overhead and increases potential candidates for *scalar replacement* by unrolling the body of a loop [AC72, KMT91a].

3.3.4 Miscellaneous transformations

Finally PED has a few miscellaneous transformations.

- **Sequential ↔ Parallel** converts a sequential DO loop into a parallel loop, and vice versa.
- **Loop bounds adjusting** adjusts the upper and lower bounds of a loop by a constant. It is used in preparation for loop fusion.
- **Statement addition** adds an assignment statement.
- **Statement deletion** deletes an assignment statement.

3.4 Transformation algorithms

The incremental update algorithms for the transformations serve a critical function; they update the code and dependence information quickly and immediately, allowing users to understand the changes, see the effects, and continue the transformation process without reanalyzing the entire program. Although many of the algorithms for applying these transformations have appeared elsewhere, our implementation gives profitability advice and performs incremental updates of dependence information. Rather than describe all these phases for each transformation, we have chosen to examine only a few interesting transformations in detail. We discuss loop interchange, loop skewing, loop distribution, and unroll and jam. The purpose, mechanics, and safety of these transformations are presented, followed by their profitability estimates, user advice, and incremental dependence update algorithms.

3.4.1 Loop interchange

Loop interchange has been used extensively in vectorizing and parallelizing compilers to adjust the granularity of parallel loops and to expose parallelism [AK87, KKLW84, Wol86]. PED interchanges pairs of adjacent loops. Loop permutations may be performed as a series of pairwise interchanges. PED supports interchange of triangular or skewed loops. It also interchanges hexagonal loops that result after skewed loops are interchanged.

Safety

Loop interchange is safe if it does not reverse the order of execution of the source and sink of any dependence. PED determines this by examining the direction vectors for all dependences carried on the outer loop. If any dependence has a direction vector of the form $(<,>)$, interchange is unsafe. These dependences are called *interchange preventing*. They are precomputed and recorded in a flag in the dependence edge. Each dependence edge carried on the outer loop is examined. If any one of these has the interchange preventing flag set, PED advises the user that interchange is unsafe.

Profitability

PED judges the profitability of loop interchange by calculating which of the loops will be parallel after the interchange. A dependence carried on the outer loop will move inward if it has a direction vector of the form $(<,=)$. These dependences are called *interchange sensitive*. They are also precomputed and stored in a flag on each dependence edge. PED examines each dependence edge on the outer loop to determine where it will be following interchange. It then checks for dependences carried on the inner loop as well; they move outward following interchange. Depending on the result, PED advises the user that neither, one, or both of the loops will be parallel after interchange.

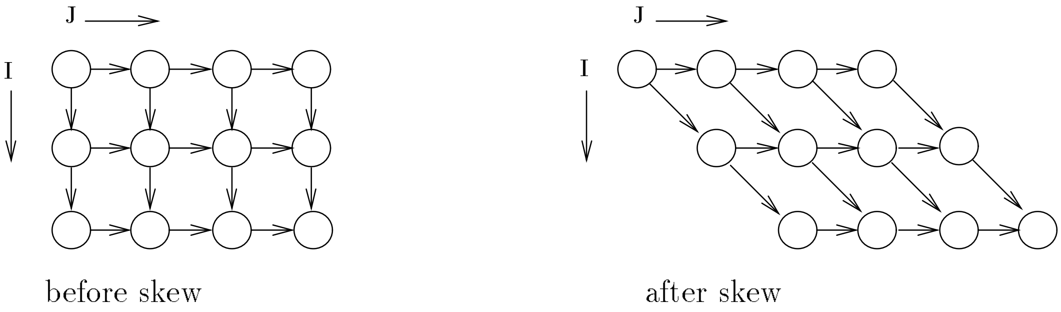
Update

Updates after loop interchange are very quick. Dependence edges on the interchanged loops are moved directly to the appropriate loop level based on their interchange sensitive flags. All the dependences in the loop nest then have the elements in their direction vector corresponding to the interchanged loops swapped, *e.g.,* $(<,=)$ becomes $(=,<)$. Finally, the interchange flags are recalculated for dependences in the loop nest.

3.4.2 Loop skewing

Loop skewing is a transformation that changes the shape of the iteration space to expose parallelism across a wavefront [IT88, Lam74, Mur71, Wol86]. It can be applied in conjunction with loop interchange, strip mining, and loop reversal to obtain

FIGURE 3.2: **Effect of loop skewing on dependences and iteration space**



effective loop-level parallelism in a loop nest [Ban90b, WL90, Wol89a]. All of these transformations are supported in PED.

Loop skewing is applied to a pair of perfectly nested loops that both carry dependences, even after loop interchange. Loop skewing adjusts the iteration space of these loops by shifting the work per iteration, changing the shape of the iteration space from a rectangle to a parallelogram, as illustrated in Figure 3.2. Skewing changes dependence distances for the inner loop so that all dependences are carried on the outer loop after loop interchange. The inner loop can then be safely parallelized.

Loop skewing of degree α is performed by adding α times the outer loop index variable to the upper and lower bounds of the inner loop, followed by subtracting the same amount from each occurrence of the inner loop index variable in the loop body. In the example below, the loop nest on the right results when the J loop in the left loop nest is skewed by degree 1 with respect to loop I.

```
DO I = 1, 100
    DO J = 2, 100
        A(I,J) = A(I-1,J) + A(I,J-1)
    ENDDO
ENDDO
```

```
DO I = 1, 100
    DO J = I + 2, I + 100
        A(I,J-I) = A(I-1,J-I) + A(I,J-I-1)
    ENDDO
ENDDO
```

Figure 3.2 illustrates the iteration space for this example. For the original loop, dependences with distance vectors $(1, 0)$ and $(0, 1)$ prevent either loop from being safely parallelized. In the skewed loop, the distance vectors for dependences are transformed to $(1, 1)$ and $(0, 1)$. There are no longer any dependences within each column of the iteration space, so parallelism is exposed. However, to introduce the parallelism on the I loop requires a loop interchange.

Safety

Loop skewing is always safe because it does not change the order in which array memory locations are accessed. It only changes the shape of the iteration space.

Profitability

To determine if skewing is profitable, PED ascertains whether skewing will expose parallelism that can be made explicit using loop interchange and suggests the minimum skew amount needed to do so. This analysis requires that all dependences carried on the outer loop have precise distance vectors. Skewing is only profitable if:

1. $\exists$ dependences on the inner loop, and
2. $\exists$ at least one dependence on the outer loop with a distance vector (d_1, d_2), where $d_2 \leq 0$.

The interchange preventing or interchange sensitive dependences in case (2) prevent the application of loop interchange to move all dependences to the outer loop. If they do not exist, at least one loop may already be safely parallelized, possibly by using loop interchange. The purpose of loop skewing is to change the distance vector to (d_1, d_2'), where $d_2' \geq 1$. In terms of the iteration space, loop skewing is needed to transform dependences that point down or downwards to the left into dependences that point downwards to the right. Followed by loop interchange, these dependences will remain on the outer loop, allowing the inner loop to be safely parallelized.

To compute the skew degree, we first consider the effect of loop skewing on each dependence. When skewing the inner loop with respect to the outer loop by an integer degree α, the original distance vector (d_1, d_2) becomes $(d_1, \alpha d_1 + d_2)$. So for any dependence where $d_2 \leq 0$, we want α such that $\alpha d_1 + d_2 \geq 1$. To find the minimal skew degree we compute

$$\alpha = \left\lceil \frac{1 - d_2}{d_1} \right\rceil$$

for each dependence, taking the maximum α for all the dependences; this is suggested as the skew degree.

Update

Updates after loop skewing are also very fast. After skewing by degree α, the incremental update algorithm changes the original distance vectors (d_1, d_2) for all dependences in the nest to $(d_1, \alpha d_1 + d_2)$, and then updates their interchange flags.

3.4.3 Loop distribution

Loop distribution separates independent statements inside a single loop into multiple loops with identical headers [AK87, KKP+81]. It is used to expose partial parallelism by separating statements which may be parallelized from those that must be executed sequentially. It is a cornerstone of vectorization and parallelization.

In PED the user can specify whether distribution is for the purpose of vectorization or parallelization. If the user specifies vectorization, then each statement is placed in a separate loop when possible. If the user specifies parallelization, then statements are grouped together into the fewest loops such that the most statements can be made parallel and the original statement order is maintained. The user is presented with a partition of the statements into new loops, as well as an indication of which loops are parallelizable. The user may then apply or reject the distribution partition.

Safety

To maintain the meaning of the original loop, the partition must not put statements that are involved in *recurrences* into different loops [KM90, KKP+81]. Recurrences are calculated by finding strongly connected regions in the subgraph composed of loop-independent dependences and dependences carried on the loop to be distributed. Statements not involved in recurrences may be placed together or in separate loops, but the order of the resulting loops must preserve all other data and control dependences. PED always computes a partition which meets these criteria.

If there is control flow in the original loop, the partition may cause decisions that occur in one loop to be used in a later loop. These decisions correspond to loop-independent control dependences that cross between partitions. We use the method described in Section 4.2 to insert new arrays, called *execution variables*, that record these "crossing" decisions. Given a partition, this algorithm introduces the minimal number of execution variables necessary to effect the partition, even for loops with arbitrary control flow.

Profitability

Currently PED does not change the order of statements in the loop during partitioning. This simplification improves the recognizability of the resulting program, but may reduce the parallelism uncovered. In particular, statements that fall lexically between statements in a recurrence will be put into the same partition as the recurrence. In addition, when the source of a dependence lexically follows the sink, these statements will be placed in the same partition. A more flexible partitioning algorithm that allows statements to be reordered is described in Section7.3.

When distributing for vectorization, statements not involved in recurrences are placed in separate loops. When distributing for parallelization, they are partitioned as follows. A statement is added to the preceding partition only if it does not cause that partition to be sequentialized. Otherwise it begins a new partition. Consider distributing the loop on the left for parallelization.

```
      DO I = 2, N                    PARALLEL DO I = 2, N
S1       A(I) = ...            S1       A(I) = ...
S2       ... = A(I - 1)              ENDDO
      ENDDO                          PARALLEL DO I = 2, N
                               S2       ... = A(I - 1)
                                     ENDDO
```

This loop contains only the loop-carried true dependence $S_1 \delta S_2$. Since there are no recurrences, S_1 and S_2 begin in separate partitions. S_1 is placed in a parallel partition, then S_2 is considered. The addition of S_2 to the partition would instantiate the loop-carried true dependence, causing the partition to be sequential. Therefore, S_2 is placed in a separate loop and both loops may be made parallel as seen on the right above.

Update

Updates can be performed quickly on the existing dependence graph after loop distribution. Data and control dependences between statements in the same partition remain unchanged. Data dependences between statements placed in separate partitions are converted from loop-carried dependences into loop-independent dependences (as in the above example).

Loop-independent control dependences that cross partitions are deleted and replaced as follows. First, loop-independent data dependences are introduced between the definitions and uses of execution variables representing the crossing decision. A control dependence is then inserted from the test on the execution variable to the sink of the original control dependence. The update algorithm is explained more thoroughly in Section 4.2.

3.4.4 Unroll and jam

Unroll and jam is a transformation that *unrolls* an outer loop in a loop nest, then *jams* (or *fuses*) the resulting inner loops [AC72, CCK88]. Unroll and jam can be used to convert dependences carried by the outer loop into loop independent dependences or dependences carried by some inner loop. It brings two accesses to the same memory location closer together and can significantly improve performance by enabling reuse of either registers or cache. When applied in conjunction with *scalar replacement* on scientific codes, unroll and jam has resulted in integer factor speedups, even for single processors [CCK90]. Unroll and jam may also be applied to imperfectly nested loops or loops with complex iteration spaces. Figure 3.3 shows an example iteration space before and after unroll and jam of degree 1.

Before performing unroll and jam of degree α on a loop with step σ, we may need to use *loop splitting* to make the total number of iterations divisible by $\alpha + 1$ by separating the first few iterations of the loop into a preloop. We then create α additional copies of the loop body. All occurrences of the loop index variable in the I^{th} new loop body must be incremented by $\sigma\mathrm{I}$. The step of the loop is then increased to $\sigma(\alpha + 1)$. Consider the following.

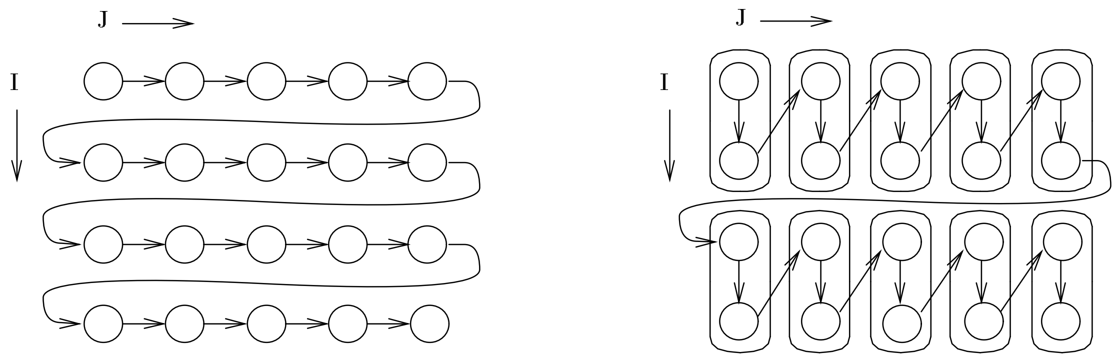
before:

```
DO I = 1, 100
    DO J = 1, 100
        C(I, J) = 0.0
        DO K = 1, 100
            C(I, J) = C(I, J) + A(I, K) * B(K, J)
        ENDDO
    ENDDO
ENDDO
```

after:

```
DO I = 1, 100, 2
    DO J = 1, 100
        C(I, J) = 0.0
        C(I + 1, J) = 0.0
        DO K = 1, 100
            C(I, J) = C(I, J) + A(I, K) * B(K, J)
            C(I + 1, J) = C(I + 1, J) + A(I + 1, K) * B(K, J)
        ENDDO
    ENDDO
ENDDO
```

FIGURE 3.3: **Effect of unroll and jam on iteration space**



before unroll and jam

after unroll and jam

In the above matrix multiply example, loop I is unrolled and jammed by one to bring together references to B(K, J), resulting in the second loop nest. Unroll and jam may also be performed on loop J to bring together references to A(I, K).

Safety

To determine safety, an alternative formulation of unroll and jam is used. Unroll and jam is equivalent to strip mining the outer loop by the unroll degree, interchanging the strip mined loop to the innermost position, and then completely unrolling the strip mined loop. Since strip mining and loop unrolling are always safe, we only need to determine whether we can safely interchange the strip mined loop to the innermost position.

PED determines this requirement by searching for interchange preventing dependences on the outer loop. Unroll and jam is unsafe if any dependence carried by the outer loop has a direction vector of the form $(<, >)$. Even if such a dependence is found, unroll and jam is still safe if the unroll degree is less than the distance of the dependence on the outer loop, since this dependence would remain carried by the outer loop. PED will either warn the user that unroll and jam is unsafe, or provide a range of safe unroll degrees.

Unroll and jam of imperfectly nested loops changes the execution order of the imperfectly nested statements with respect to the rest of the loop body. Dependences carried on the unrolled loop with distance less than or equal to the unroll degree are converted into loop-independent dependences. If any of these dependences cross between the imperfectly nested statements and the statements in the inner loop, they inhibit unroll and jam. Specifically, the intervening statements cannot be moved and prevent fusion of the inner loops.

Profitability

Balance describes the ratio between computation and memory access rates [CCK88]. Unroll and jam is profitable if it brings the balance of a loop closer to the balance of the underlying machine. PED automatically calculates the optimal unroll and jam degree for a loop nest, including loops with complex iteration spaces [CCK90].

Update

An algorithm for the incremental update of the dependence graph after unroll and jam is described elsewhere [CCK90]. However, we chose a different strategy. Since no global data-flow or symbolic information is changed by unroll and jam, PED rebuilds the scalar dependence graph for the loop nest and refines it with dependence tests. This update strategy proved much simpler to implement and is very quick in practice.

3.5 Incremental analysis after edits

Editing is fundamental for any program development tool because it is the most flexible means of making program changes. The ParaScope Editor therefore provides advanced editing features. When editing, the user has complete access to the functionality of the hybrid text and structure editor underlying PED, including simple text entry, template-based editing, search and replace functions, intelligent and customizable view filters, and automatic syntax and type checking.

Rather than reanalyze immediately after each edit, PED waits for a reanalyze command from the user. The user may thus avoid analyzing intermediate stages of the program that may be illegal or simply uninteresting. The transformations, the dependence display and the variable display are disabled during an editing session, because they rely on dependence information that may be invalidated by the edits. Once the user prompts PED, the dependence driver invokes syntax and type checking. If errors are detected, the user is warned; otherwise, reanalysis proceeds.

Unfortunately, incremental dependence analysis after edits is a very difficult problem. Precise dependence analysis requires utilization of several different kinds of information. In order to calculate precise dependence information, PED may need to incrementally update the control flow graph, control dependence graph, static single assignment graph (SSA) [CFR+89], and call graphs, as well as recalculate scalar live range, constant, symbolic, interprocedural, and dependence testing information.

Several algorithms for performing incremental analysis are found in the literature; for example, data-flow analysis [RP88, Zad84], interprocedural analysis [Bur90, RC86], interprocedural recompilation analysis [BCKT90], as well as dependence analysis [Ros90]. However, few of these algorithms have been implemented and evaluated in an interactive environment. Rather than tackle all these problems at once, we chose a simple yet practical strategy for the current implementation of PED. First, the scope of each program change is evaluated. Incremental analysis is applied only when it may be profitable, otherwise batch dependence analysis is invoked. PED will apply incremental dependence analysis when the following situations are detected:

No update needed

Many program edits fall into this category. It is trivial using a structure editor to determine that changes to comments or whitespace do not require reanalysis. Other more interesting cases include changes to arithmetic expressions that do not disturb control flow or symbolic analysis. For instance, changing the assignment A(I) = B(I) to A(I) = B(I) + 1 does not affect dependence information one whit.

Delete dependence edges

Removal of an array reference may be handled simply by deleting all edges involving that reference.

Add dependence edges

Addition of an array reference may be handled by scanning the loop nest for occurrences of the same variable, performing dependence tests between the new reference and any other references, and adding the necessary dependence edges.

Redo dependence testing

Changes to loop bounds or array subscript expressions require dependence testing to be performed on all affected array variables.

Redo local symbolic analysis

Some types of program changes do not affect the scalar dependence graph, but may require symbolic analysis to be reapplied. For instance, changing the assignment J=J+1 to J=J+2, where J is an auxiliary induction variable, requires redoing symbolic analysis and dependence testing.

Redo local dependence analysis

Changes such as the modification of control flow or variables involved in symbolic analysis require significant updates best handled by redoing dependence analysis. However, the nature of the change may allow the reanalysis to be limited to the current loop nest or procedure. In these cases, the entire program does not need to be reanalyzed.

3.6 User and compiler interaction

Once the programmer begins changing the source for the purpose of optimization, version control begins to play an important role. Is the new version now machine dependent, or is it a better machine independent version, or it is a bug fix? The automation of version control is still an open question and in the current implementation, version control is left to the programmer. In order to facilitate a single machine-independent program version we propose the following compiler-controlled approach.

In this approach, the compiler would first perform automatic parallelization as a source-to-source transformation producing a machine specific version. If the user is satisfied with the program's resulting performance, the user need not intervene at all. If the user is unsatisfied, the compiler communicates to the user in the interactive tool, in this case PED. The compiler would mark the loops in the original version that it was unable to parallelize or parallelize well. It would also rank the loops and subroutines by their effects on execution time using performance estimation or run-time profiling. If the user wants to maintain portability, the onus shifts to the user to make assertions and improvements in an architecture independent manner. The user is assisted with the hints and functionality currently provided PED, such as dependence display and transformations. In addition, users would be able to invoke the compiler's optimizing and parallelizing algorithms in PED to determine the effects of their changes, providing almost an interactive compiler.

3.7 Related work

Several other research groups are also developing advanced parallel programming tools. PED's analysis and transformation capabilities compare favorably to automatic parallelization systems such as Parafrase, PTRAN, and of course PFC. Our work on interactive parallelization bears similarities to SIGMACS, PAT, and SUPERB.

PED has been greatly influenced by the Rice Parallel Fortran Converter (PFC), which has focused on the problem of automatically vectorizing and parallelizing sequential Fortran [AK87]. PFC has a mature dependence analyzer which performs data dependence analysis, control dependence analysis, interprocedural constant propagation [CCKT86], interprocedural side-effect analysis of scalars [CKT86a], and interprocedural array section analysis [CK87b, HK91]. PED expands on PFC's analysis and transformation capabilities and makes them available to the user in an interactive environment. Because of its mature analysis and implementation, PFC is available as a dependence information server for the ParaScope Editor. On demand, the information provided by PFC is converted into the internal representations in ParaScope Editor. This functionality enables the use of PFC's more advanced analysis in PED.

Parafrase was the first automatic vectorizing compiler [KKLW84]. It supports program analysis and performs a large number of program transformations to improve parallelism. In Parafrase, program transformations are structured in phases and are always applied where applicable. Batch analysis is performed after each transformation phase to update the dependence information for the entire program. Parafrase-2 adds scheduling and improved program analysis and transformations [PGH+90]. More advanced interprocedural and symbolic analysis is planned [HP90]. Parafrase-2 uses FAUST as a front end to provide interactive parallelization and graphical displays [GGGJ88].

PTRAN is also an automatic parallelizer with extensive program analysis. It computes the SSA and *program dependence graphs*, and performs constant propagation and interprocedural analysis [CFR+89, FOW87]. PTRAN introduces both task and loop parallelism, but the only other program transformations are variable privatization and loop distribution [ABC+87, Sar90].

SIGMACS, a programmable interactive parallelizer in the FAUST programming environment, computes and displays call graphs, process graphs, and a statement dependence graph [GGGJ88, SG90]. In a process graph each node represents a task or a process, which is a separate entity running in parallel. The call and process graphs may be animated dynamically at run time. SIGMACS also performs several interactive program transformations, and is planning on incorporating automatic updates of dependence information.

PAT is also an interactive parallelization tool [SA88, SA89]. Its dependence analysis is restricted to Fortran programs where only one write occurs to each variable in a loop. In addition, PAT uses simple dependence tests that do not calculate general distance or direction vectors. Hence, it is incapable of applying loop level transformations such as loop interchange and skewing. However, PAT does support replication and alignment, insertion and deletion of assignment statements, and loop parallelization for a single loop. It can also insert synchronization to protect specific dependences. PAT divides analysis into scalar and dependence phases, but does not perform symbolic or interprocedural analysis. The incremental dependence update that follows transformations is simplified due to its austere analysis [SAS90].

SUPERB interactively converts sequential programs into data parallel SPMD programs that can be executed on the SUPRENUM distributed memory multiprocessor [ZBG88]. SUPERB provides a set of interactive program transformations, including transformations that exploit data parallelism. The user specifies a data partitioning, then node programs with the necessary *send* and *receive* operations are automatically generated. Algorithms are also described for incremental update of use-def and def-use chains following structured program transformations [KZBG88].

3.8 Discussion

The ParaScope Editor provides a complementary strategy to backup automatic parallelization. In an integrated approach that makes the compiler algorithms available, as well as the individual transformations, the user may make assertions and see the results in the automatically generated version. It also enables users to experiment with different mixtures of transformations without reanalyzing the entire program between transformations.

Our experience with the ParaScope Editor has shown that dependence analysis can be used in an interactive tool with ample efficiency [HHK+93]. This efficiency is due to fast yet precise dependence analysis algorithms, and a dependence representation that makes it easy to find dependences and to reconstruct them after a change. To our knowledge, PED is the first tool to offer general editing with dependence reconstruction along with a substantial collection of useful program transformations.

Chapter 4

Loop Transformations with Arbitrary Control Flow

Previous code generation techniques and program transformations have known limitation dealing with control flow. Many of these transformations are loop based and are not applicable when there exists control flow such as branching within a loop or exit branches out of a loop. Because most programs contain meaningful control flow in loops, this limitation is a serious flaw. For truly effective parallel code generation, this problem must be addressed.

In this chapter, we extend a broad selection of transformations from Section 3.3 to deal with arbitrary control flow, thus allowing an integrated transformation system for parallel code generation or interactive parallelization that is not inhibited by control flow. Several transformations are inhibited by some type of control flow and others are easily extended. For example, loop permutation is safe when branches are internal and is inhibited by exit branches. Strip mining, on the other hand, is safe regardless of the type of branching. Two particularly important transformations, loop distribution and loop fusion require more sophisticated algorithms that leverage the control dependence graph.

In the next section, we give a motivating example, an introduction to the problems with previous work, a few definitions and our general approach. Due to its wide spread use and the new and optimal results presented here, We begin with the algorithm for loop distribution when loops contain arbitrary control flow. The following section detail a variety of other transformations and includes loop skewing, loop reversal, loop permutation, strip mining, privatization, scalar expansion, loop fusion and loop peeling.

4.1 Motivation

To motivate our treatment of conditional control flow, we first consider loop distribution. Loop distribution breaks up a single loop into two or more loops, each of which iterates over a disjoint subset of the statements in the body of the original loop. The usefulness of this transformation derives from its ability to convert a large loop whose iterations cannot be run in parallel into multiple loops, many of which can be parallelized. Consider the following code.

```
DO I = 2, N
      A(I) = B(I) + C
      D(I) = A(I-1)*E
ENDDO
```

If we wish to retain the original meaning of this code fragment, the iterations cannot be run in parallel without explicit synchronization lest a value of A(I-1) is fetched before the previous iteration has a chance to store it. However, if the loop is distributed, each of the resulting loops can be run in parallel.

```
DOALL I = 2, N
      A(I) = B(I) + C
ENDDO
DOALL I = 2, N
      D(I) = A(I-1)*E
ENDDO
```

The presence of conditionals complicates distribution. Consider, for example the following loop.

```
DO I = 2, N
      IF (A(I) .EQ. 0) THEN
            A(I) = B(I) + C
            D(I) = A(I-1)*E
      ENDIF
ENDDO
```

In order to place the first assignment in the first loop and the second assignment in the second loop, the result of the IF statement must be known in both loops. The IF cannot be replicated in both loops, because the first assignment changes the value of A. One solution to this problem is to convert all IF statements to conditional assignment statements, as follows:

```
DO I = 2, N
      P(I) = A(I) .EQ. 0
      IF (P(I)) A(I) = B(I) + C
      IF (P(I)) D(I) = A(I-1)*E
ENDDO
```

The resulting loop can be distributed by considering only data dependence, because the control dependence has been converted to a data dependence involving the logical array P. This approach, called *if-conversion* [AKPW83, All83], has been used successfully in a variety of vectorization systems which incorporate several other transformations as well [AK87, SK86, KKLW84]. However, if-conversion has several drawbacks. If vectorization or parallelization fails, it is not easy to reconstruct efficient branching code. In addition, if-conversion may cause significant increases in the code space to hold conditionals.

For these reasons, research in automatic parallelization has concentrated on an alternative approach that uses control dependences to model control flow [FOW87, ABC+87, ABC+88]. Our approach uses both data and control dependence graphs (as were defined in Sections 2.1 and 2.1). For our purposes, it is useful to classify the type of control flow in a loop nest and its correspondence in the control dependence graph as either

1. *internal branching* or
2. *exit branching.*

Internal branching consists of conditional control flow that affects only the statements executed on a particular iteration of the loop. These are loop-independent control dependences. Exit branching is conditional control flow which terminates the execution of the loop. These are loop-carried control dependences. Internal branching may utilize structured or unstructured constructs. Exit branching can only be formed using unstructured control flow (GOTOs in Fortran).

Exit branches are inherently sequential because they give rise to loop carried dependences. Although the main focus of this dissertation is the use of transformations in a parallelizing environment, many of the transformations below are also useful for scalar compilation and data locality optimizations. Therefore, the extensions and limitations necessary for both internal and exit branching are include in the discussion below.

4.2 Loop distribution

Loop distribution is an integral part of transforming a sequential program into a parallel one. It was introduced by Muraoka [Mur71] and is used extensively in parallelization, vectorization, and memory management. For loops with control flow, previous methods for loop distribution have significant drawbacks. We present a new algorithm for loop distribution in the presence of control flow modeled by a control dependence graph. This algorithm is shown optimal in that it generates the minimum number of new arrays and tests possible. We also present a code generation algorithm that produces code for the resulting program without replicating statements or conditions. These algorithms are very general and can be used in automatic or interactive parallelization systems.

This section presents a method for performing loop distribution in the presence of control flow based on control dependences. Control dependences may be used like data dependences for determining the placement of statements in loops. However, when there exists a control dependence between statements that crosses their new respective loop bodies, correct code generation requires recording the results of evaluating the predicate in a logical array and testing the logical array in the second loop.

Our approach is optimal in the sense that it introduces the fewest possible new logical arrays and tests. In particular, it introduces one array for each conditional node upon which some node in another loop in the distribution depends. We also present an algorithm for generating code for the body of a loop after distribution. The algorithms are very fast, both asymptotically and practically. This algorithm is also described elsewhere [KM90]

4.2.1 Mechanics

Loop distribution may be separated into a three-stage process: (1) the statements in the loop body are partitioned into groups to be placed in different output loops; (2) the control and data dependence graphs are restructured to effect the new loop organization and (3) an equivalent program is generated from the dependence graphs. To perform loop distribution without changing the original meaning of the loop, the placement of statements into new loops must preserve the data and control dependences of the original. The method we present is designed to work on any partition that is *legal, i.e.*, any partition that preserves the control and data dependences.

A partition can preserve all dependences if and only if there exists no dependence cycle spanning more than one output loop [KKP+81, AK87]. If there is a cycle involving control and/or data dependences, it must be entirely contained within a single partition.[2] This condition is both necessary and sufficient. Consider what must be done to generate code given a partitioning into loops: some linear order for the loops must be chosen. If we treat each output loop as a single node and define dependence between loops to be inherited in the natural way from control and data

[2]Loops with exit branches are an exception to this condition. The necessary extensions are discussed at the end of Section 4.2.2.

dependences between statements, then the resulting graphs will be acyclic if and only if each original recurrence is confined to a single loop. Since an acyclic graph can always be ordered using topological sort and a cyclic graph can never be ordered, the condition is established.

In the algorithms presented below the nodes in both the control and data dependence graphs usually represent a single statement. Exceptions to the single statement per node rule are inner loops and irreducible regions; all of their statements are represented with a single node.

Because our algorithm accepts any legal partition as input, it is as general as possible. It can be used for vectorization, which seeks a partition of the finest possible granularity, or for MIMD parallelization, which seeks the coarsest possible granularity without sacrificing parallelism. We discuss a partitioning strategy in Section 7.3 for shared-memory parallel code generation. In the discussion here, we assume a legal partition is provided.

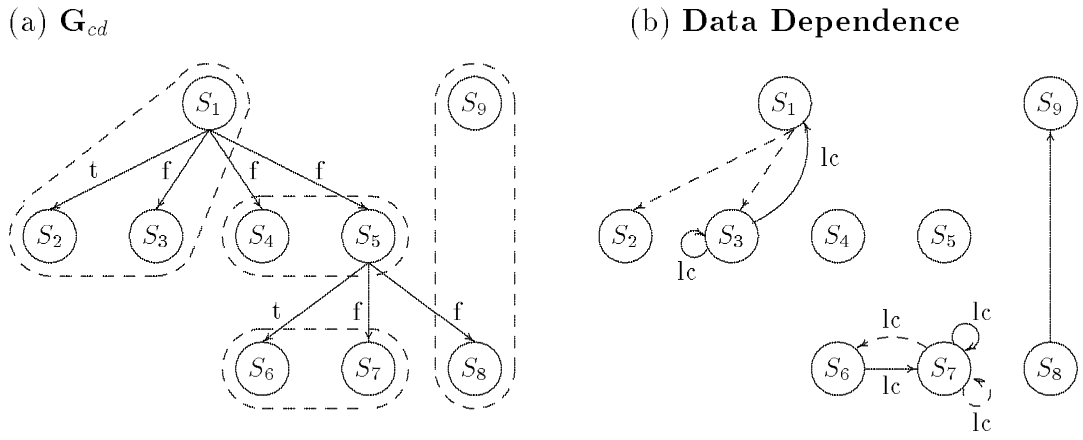
4.2.2 Restructuring

In the original program, control decisions are made and used in the same loop on the same iteration, but a partition may specify that decisions that are made in one loop be used in another. This problem is illustrated below by Example 4.1. Its corresponding G_{cd} and data dependence graph are shown in Figure 4.1.

EXAMPLE 4.1:

```
      DO I = 1, N
S1        IF (A(I) .GT. T) THEN
S2            A(I) = I
          ELSE
S3            T = T + 1
S4            F(I) = A(I)
S5            IF (B(I) .NE. 0) THEN
S6                U = A(I) / B(I)
              ELSE
S7                U = A(I) - U
S8                C(I) = B(I) + C(I)
              ENDIF
          ENDIF
S9        D(I) = D(I) + C(I)
      ENDDO
```

Figure 4.1: **Control and data dependence graphs for distribution example**



The data dependence graph in Figure 4.1(b) shows true dependences with solid lines and anti dependences with dashed lines. Loop carried edges are labeled with *lc*. In this example, output dependences are redundant and are not included. Given the data and control dependences in Figure 4.1, the statements may be placed in four partitions: (S_1, S_2, S_3), (S_4, S_5), (S_6, S_7), and (S_8, S_9). This particular partition is chosen solely for exposition of the algorithm, and in Figure 4.1(a) it is superimposed on G_{cd} such that each partition is enclosed by dashed lines.

Given this partition, some statements are no longer in the same loop with statements upon which they are control dependent. For example, S_4 is control dependent on S_1, but S_1 and S_4 are not in the same partition. In Figure 4.1 the G_{cd} edges that cross partitions represent decisions made in one loop, and used in a later loop. There may be a chain of decisions on which a node n is control dependent, but given a legal partition, all of n's immediate predecessors and ancestors in G_{cd} are guaranteed either to be in n's partition, or in an earlier one. Therefore the execution of n may be determined solely from the execution of n's predecessors. We introduce *execution variables* to compute and store decisions that cross partitions in G_{cd} for both structured and unstructured code.

Execution variables

Execution variables are only needed for *branch nodes*, because they correspond to control decisions in the original program. Any node in G_{cd} that has a successor must be a branch node, but only branch nodes with at least one successor in a different partition are of interest here. For each branch in this restricted set, a unique execution variable is created. Only one execution variable is created, regardless of the number of successors or the number of different partitions to which the successors belong. The execution variable is assigned the value of the test at the branch, capturing the branch decision. Later this variable will be tested to determine control flow in a subsequent partition. Hence, the creation of an execution variable will replace control dependences between partitions with data dependences. Execution variables are arrays, with one value for each iteration of the loop, because each iteration can give rise to a different control decision. If desired, loop invariant decisions can be detected [AC72] and represented with scalar execution variables.

All previous techniques, whether they are G_{cd} based or not, use Boolean logic when introducing arrays to record branch decisions. These methods require either testing and recording the path taken in previous loops or introducing additional arrays. In Example 4.1 in the loop with statements (S_6, S_7), either S_6, or S_7, or neither may execute on a given iteration. Because there are three possibilities, the correct decision cannot be made with a single Boolean variable. For example, if S_1 takes the true branch, then neither S_6 nor S_7 should execute. If just S_5's decision is stored, then one of S_6 or S_7 will mistakenly be executed, because the branch recording array for S_5 must either be true or false, regardless of S_1's decision.

Given this drawback, we have formulated execution variables to have three possible values: *true*, *false* and $\top$, which represents "undefined". Every execution variable is initialized to $\top$ at the beginning of the loop in which it will be assigned, indicating that the branch has not yet been executed. Because of the existence of a "not executed" value, the control dependent successors in different partitions need only test the value of the execution variables for their immediate predecessors; they do not need to test the entire path of their control dependence ancestors. This condition is true whether the control flow is unstructured or structured. Execution variables completely capture the control decision at a node, making them extremely powerful.

ALGORITHM 4.1: **Execution variable and guard creation**

INPUT: partitions, G_{cd}, statement order
OUTPUT: modified G_{cd} with execution variables
ALGORITHM:

> **for** each partition, P
>> **for** each $n \in P$, in order
>>> **if** ($\exists$ an edge $(n, o)_l \in G_{cd}$, where $o \notin P$)
>>>> insert "EV$_n$(I) = $\top$" into P at top
>>>> let *test* be n's branch condition
>>>> **if** ($\exists$ $(n, m)_l$ where $m \in P$)
>>>>> replace n with $\begin{cases} \text{"EV}_n\text{(I) = test"} \\ \text{"IF (EV}_n\text{(I) .EQ. TRUE)"} \end{cases}$
>>>> **else**
>>>>> replace n WITH "EV$_n$(I) = test"
>>>> **endif**
>>>> **for** each $P_k \neq P$ containing a successor of n
>>>> (Build guards, and modify G_{cd})
>>>>> **for** each l where $\exists$ $(n, p)_l$ with $p \in P_k$
>>>>>> create new statement N:
>>>>>> "IF (EV$_n$(I) .EQ. l)",
>>>>>> add N to P_k (N is new and unique)
>>>>>> insert data dependences for EV$_n$
>>>>>> **for** each $(n, q)_l$ where $q \in P_k$
>>>>>> (Update control dependences)
>>>>>>> delete $(n, q)_l$ from G_{cd}
>>>>>>> add $(N, q)_{true}$ to G_{cd}
>>>>>> **endfor**
>>>>> **endfor**
>>>> **endfor**
>>> **endif**
>> **endfor**
> **endfor**

Restructuring

The restructuring Algorithm 4.1 creates and inserts execution variables and guards, given a distribution partition. It also updates the control and data dependence graphs to reflect the changes it makes. The algorithm is applied in partition order and, within a partition, in statement order over G_{cd} (statement order is the original lexical order). The algorithm can be subdivided into three parts. First, execution variables for a branch node n are created where needed. Next, guard expressions are inserted for any nodes control dependent on n. Then the control and data dependences are updated, reflecting the new guards and execution variables.

The need for an execution variable for n is determined by considering n's immediate successors. If there is an outgoing edge from n to a node that is not in n's partition, an execution variable is created. In Example 4.1, execution variables are needed for S_1 and S_5. The initialization of the execution variable is inserted at the beginning of n's partition, ensuring it will always be executed. Next, an assignment of the execution variable to n's test is inserted in node n. If n has successors in its partition, its branch is changed to test the execution variable. Otherwise, its branch is deleted.

For each partition P_k that contains a successor of n, a guard on n's execution variable is built. Here the successors of n are also considered in statement order. A guard is built for every distinct label from n into P_k. Each guard compares n's execution variable, $\text{EV}_n(\text{I})$, to the distinct label l. All of n's successors in G_{cd} in P_k on label l are severed from n and connected to the newly created corresponding guard. Our examples have only two labels, true and false, but any number of branch targets can be handled.

Consider Example 4.1. S_5 has successors in two partitions, (S_6, S_7) and (S_8, S_9). The successors in (S_6, S_7) are on different branches. S_6 is on the true branch, so the guard expression created is "$\text{EV}_5(\text{I})$ .EQ. TRUE." S_7 is on the false branch, so its guard expression is "$\text{EV}_5(\text{I})$ .EQ. FALSE." The old edges (5, 6) and (5, 7) are deleted from G_{cd}, and new edges attaching 6 and 7 to their corresponding guards are created. Similarly a guard is created for and connected to S_8.

The following simple optimization is included in the algorithm and examples but, for clarity, does not appear in the statement of the algorithm. Determining whether the initialization of an execution variable is necessary can be accomplished when an execution variable is created for a node n. If n is not control dependent on any other node, *i.e.*, a root in the control dependence graph, then there is no need for initialization to be inserted. During guard creation for the successors of this node, the execution variable is known to have a value other than $\top$. Therefore, if control flow is structured, only one guard is needed for each successor partition instead of for each label.

After restructuring is applied, each partition has a correct G_{cd}, a correct data dependence graph, and possibly some new statements (execution variable assignments and guards). At this point the code for the distribution partition can be generated. We use a simple code generation algorithm, which is described in Section 4.2.3. Given the distribution in Figure 4.1 for Example 4.1, restructuring and code generation results in the following code.

```
      DO I = 1, N
           EV1(I) = A(I) .GT. T
S1         IF (EV1(I) .EQ TRUE) THEN
S2              A(I) = I
           ELSE
S3              T = T + 1
           ENDIF
      ENDDO
      DO I = 1, N
           EV5(I) = ⊤
           IF (EV1(I) .EQ. FALSE) THEN
S4              F(I) = A(I)
S5              EV5(I) = B(I) .EQ. 0
           ENDIF
      ENDDO
      DO I = 1, N
S6         IF (EV5(I) .EQ. TRUE) U = A(I) / B(I)
S7         ELSE IF (EV5(I) .EQ. FALSE) U = A(I) - U
      ENDDO
      DO I = 1, N
S8         IF (EV5(I) .EQ. FALSE) C(I) = B(I) + C(I)
S9         D(I) = D(I) + C(I)
      ENDDO
```

The advantages of three-valued logic are illustrated by the concise guards for S_6 and S_7. As shown in Section 4.2.2, $\text{EV}_5(\text{I})$ must be explicitly tested for true or false, because if S_1 evaluated to true, then $\text{EV}_5(\text{I})$ will be $\top$ and neither S_6 nor S_7 should execute. Not only do we avoid testing $\text{EV}_1(\text{I})$ here, if S_4 and S_5 were in S_1's partition, there would be no need to store S_1's decision at all, even though S_6 are S_7 *indirectly* dependent on S_1 and S_1 remains in a different loop.

Optimality

Given a distribution, this section proves that our algorithm creates the minimal number of execution variables needed to track control decisions affecting statement execution in other loops. It also establishes that the algorithm produces the minimal number of guards on the values of an execution variable required to correctly execute the distributed code. Therefore, our algorithm is optimal for a given distribution partition.

Lemma 4.1 Each execution variable represents a unique decision that must be communicated between two loops.

Proof. An execution variable is created only when a decision in one partition directly affects the execution of a statement in another partition, as specified by G_{cd}. The definition of G_{cd} guarantees that no decision node subsumes another, and therefore any decisions represented by execution variables are unique. □

The restructuring algorithm creates the minimal number of guards on the values of an execution variable required to correctly determine execution. Let

p = the number of distinct partitions, P, and

m = the number of distinct branch labels, l,

that contain successors of node n. There are at most k tests on the value of an execution variable EV_n, where

$$k = \sum_{i=1}^{p} \sum_{j=1}^{m} (l_j \in P_i).$$

k is the sum of distinct labels into every distinct partition, and is bounded by the number of n's successors that are in separate partitions P_i.

Theorem 4.1 The number of guards that test an execution variable is the minimum required to preserve correctness for the given distribution.

Proof. By contradiction. If there exists a version of the distribution with fewer guards, then guards would be produced that were either unnecessary or redundant. If there were unnecessary guards, then Lemma 4.1 would be violated. If there were redundant guards, then there would be multiple guards for nodes in the same partition with the same label. However, the algorithm produces at most one guard per label used in a partition. □

Exit branches

Because exit branches determine the number of iterations that are executed for an entire loop, they are somewhat sequential in nature. It is possible to perform distribution on such loops in a limited form by placing all exit branches in the first partition. Of course any other statements involved in recurrences with these statements must also be in the first partition. This forces the number of iterations to be completely determined by the first loop. If there are any statements left, any legal partitioning of them may be performed. The control dependences for each of the subsequent partitions can be satisfied with execution variables as described above. However, during code generation their loop bounds must be adjusted. If an exit branch was taken, any statements preceding it in the original loop must execute the same number of times as the first loop, later statements must execute one less time than the first loop. Otherwise, when no exit branch is taken, all loops must execute the same number of times as the first loop.

4.2.3 Code generation

To review, there are three phases to distribution in the presence of control flow. The first step determines a partitioning based on data and control dependences. The second step inserts execution variables and guards to effect the partition and updates the control and data dependences. The third step is code generation.

In step two the only changes to the data dependence graph are the addition of edges that connect the definitions of execution variables to their uses. A G_{cd} is built for each new loop during this phase. In each new loop's G_{cd} there are no control dependences between guards. However, there may be relationships between execution variables that can be exploited during code generation.

Now we consider code generation for unstructured or structured control flow without exit branches (Section 4.2.2 outlines the extensions for exit branches). Because the data and control dependence graphs, as well as the program statements are correct on entry to the code generation phase, a variety of code generation algorithms could be used. For example, any of the code generation algorithms based on the *program dependence graph* could be used in conjunction with the above algorithm [FM85, FMS88, CFS90, BB89]. The very simple code generation scheme described here has been is implemented in the ParaScope Editor [KMT91a].

When transformations are applied in an interactive environment, it is important to retain as much similarity to the original program as possible. The programmer can more easily recognize and understand transformed code when it resembles the original. For this reason, although partitioning may cause statements to change order, the original statement order and control structure *within* a partition is maintained. If the original loop is structured, the resulting code will be structured. If the original loop was unstructured and difficult to understand, so most likely will be the distributed loop.

To maintain the original statement ordering, an ordering number is computed and stored in order[n]. All the nodes in G_{cd} are numbered relative to their original lexical order, from one to the number of nodes. All of the execution variable initialization nodes are numbered zero, so they will always be generated before any other node in their partition. The newly created guard nodes have an order number and a relative number, rel[n]. Their order numbers are the number of the node whose execution variable appears in the guard expression. Their relative numbers, rel[n], are the number of the guard's lowest numbered successor. Both of these numbers can be computed when the guard is created. To simplify the discussion, branches are assumed to have only two label values, true and false, but the algorithm may be easily extended for multi-valued branches.

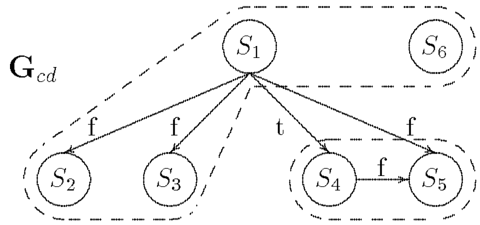
The rest of our discussion is divided into three parts. First relabeling, which corrects and renames statement labels, is described. The code generation discussion is separated into one section for structured and one for unstructured code.

Label renaming

A distribution partition may specify that the destination of a GOTO, that is, a labeled statement, be in a different loop from the GOTO. Replication and label renaming of GOTOs of this type must be performed to compensate for this after restructuring and before code generation. Renaming is easily accomplished by replacing the destination of a GOTO that is no longer in the same loop with an existing label or a new label, l_{P_j}, which may require a CONTINUE. The new destination has the same relative ordering as the original label. Often this will be the last statement in the partition. Reuse of labels is done whenever possible.

EXAMPLE 4.2:

```
     DO I = 1, N
S1      IF (p1) GOTO 4
        S2
S3      GOTO 5
4       IF (p4) GOTO 6
5       S5
6       S6
     ENDDO
```



Consider Example 4.2 with a distribution partition (S_1, S_2, S_3, S_6) and (S_4, S_5). The destination of S_1's GOTO, S_4, is not in the same partition as S_1, therefore the GOTO's label must be renamed. In this case, the new destination of S_1's jump must not interfere with the execution of S_6. To determine the destination and new label, the statement number of the original labeled statement (in this case 4) is compared to each statement in the partition following S_1 in order. When a statement number greater than the original is found (S_6 in our example), its label is used or a new one is created for it. Any empty jumps are deleted. A straightforward relabeling of the first partition in Example 4.2 after restructuring results in the following.

```
      DO I = 1, N
            EV1[I] = p1
S1          IF (EV1[I] .EQ. TRUE) GOTO 6
            S2
      6     S6
      ENDDO
```

Structured code generation

When G_{cd} is a tree, code generation is relatively simple [FM85, FMS88, BB89]. This discussion emphasizes properly selecting and inserting the appropriate control structures for newly created guards. Other G_{cd} code generation algorithms must select and create control structures for all branches. Because we use the original control structures for all but the newly created guards, only they are of interest here. When the guards are created, they are identified by setting *guard* $[n]$ to true. For all other nodes, *guard* $[n]$ evaluates to false. With structured control flow the only two control structures that need be inserted when generating guards are IF-THEN and IF-THEN-ELSE.

Our algorithm for code generation from structured or unstructured code appears in Figure 4.2. It considers each partition and its nodes based on their order number, from lowest to highest. If a node n is not a guard node, it is generated with its original control structure followed by any descendants using depth-first recursion on G_{cd}. Given a tree G_{cd}, and that all control dependences are satisfied, the ancestors of a node n in G_{cd} must be generated before n is. If the node is a guard node, the control structure for it must be selected and created. This work is done in the procedure *genguard*.

ALGORITHM 4.2: Code generation after distribution

Codegen(n)

INPUT: n is a statement node
G$_{cd}$, ordered partitions, order[n], rel[n], guard[n], goto[n]

OUTPUT: The distributed loops

ALGORITHM:

```
for each partition, P
    gen (DO) (The original loop header)
    while (∃ n ∈ P)
        choose n with smallest order[n] and
            if goto[n] and not only predecessor,
            with greatest rel[n], otherwise smallest rel[n]
        done = false
        delete n from P
        if (guard[n])
            genguard (n)
        else
            gen (n)
            gensuccessors (n, all) (all matches any branch label)
        endif
    endwhile
endfor
```

gensuccessors (n, l)

INPUT: n is a statement node
l is a label

ALGORITHM:

```
while (done = false and ∃ (n, m)_l ∈ G_cd and P)
    choose m with smallest order[m]
    if (∃ (p, m) where p ≠ n)
    (In structured code m has one predecessor and this will never occur)
        done = true
    else
        delete (m) from P
        gen (m)
        gensuccessors (m, all)
    endif
endwhile
```

ALGORITHM 4.2
Continued

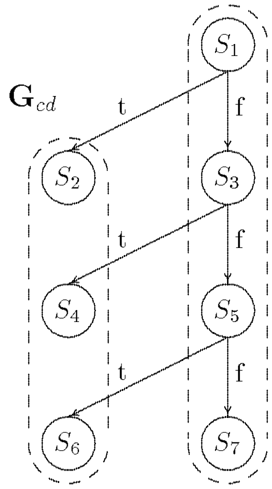
```
genguard (n)
    INPUT:        n is a statement node
    ALGORITHM:
      (Generate unstructured code)
        if (∃ (p, rel[n]) p ≠ n and p still ∈ P)
              let L be the statement label of node rel[n]
              gen (IF n GOTO L)

      (Generate structured constructs)
        (The original conditional was structured)
        else if (∃ (n, q)_true and (n, r)_false where order[q] < order[r])
              gen (IF n THEN)
              gensuccessors (n, true)
              gen (ELSE)
              gensuccessors (n, false)
              gen (ENDIF)

        (The original conditional was structured)
        else if (∃ o where order[o] = order[n])
              (n chosen s.t. rel[n] < rel[o])
              gen (IF n THEN)
              gensuccessors (n, true)
              delete o from P
              gen (ELSE IF o THEN)
              gensuccessors (o, true)
              gen (ENDIF)

        (The original could be unstructured or structured)
        else
              gen (IF n THEN)
              gensuccessors (n, true)
              gen (ENDIF)
        endif
    end
```

If the guard node has true and false branches, an IF-THEN-ELSE is generated, where the conditional is the guard expression. For each successor on the true branch, it and its descendants are generated recursively, in order. The false successors are generated similarly under the ELSE. If there are two guards with the same order number, they are ordered by their relative number, and an IF-THEN-ELSE-IF-THEN is generated. The first guard expression becomes the first conditional, and its successors and their descendants are generated in the corresponding THEN. The second guard expression conditions the ELSE-IF-THEN, and is followed by its descendants. Otherwise the guard is the only node with this order number, and an IF-THEN is generated for the guard and its descendants.

In Example 4.3 the control dependence graph is a long narrow tree. After applying the above algorithms to this loop, the code below results. The first loop shows the dead branch optimization. The second loop illustrates that it is possible to generate correct code without adding control dependences between guards. More efficient code could be generated by noticing in the second loop nest if $\text{EV}_1[\text{I}]$ is true then neither $\text{EV}_3[\text{I}]$ or $\text{EV}_5[\text{I}]$ can be true, and similarly if $\text{EV}_3[\text{I}]$ is true then $\text{EV}_5[\text{I}]$ cannot be true. This code would not have fewer tests, but would be more efficient and have a different structure.

EXAMPLE 4.3:

```
      DO I = 1, N
S1       IF (p1) THEN
            S2
         ELSE
S3          IF (p3) THEN
               S4
            ELSE
S5             IF (p5) THEN
                  S6
               ELSE
                  S7
               ENDIF
            ENDIF
         ENDIF
      ENDDO
```



```
        DO I = 1, N
              EV3[I] = ⊤
              EV5[I] = ⊤
S1            EV1[I] = p1
              IF (EV1[I] .EQ. FALSE) THEN
S3                  EV3[I] = p3
                    IF (EV3[I] .EQ. FALSE) THEN
S5                        EV5[I] = p5
                          IF (EV5[I] .EQ. FALSE) THEN
                                S7
                          ENDIF
                    ENDIF
              ENDIF
        ENDDO
        DO I = 1, N
              IF (EV1[I] .EQ. TRUE) S2
              IF (EV3[I] .EQ. TRUE) S4
              IF (EV5[I] .EQ. TRUE) S6
        ENDDO
```

Unstructured code generation

We can avoid the usual problems when generating code with a DAG G_{cd} for unstructured control flow by using the original structure and computing some additional information about the origin of the new guards. This information can be computed during code generation, or when the guards are created. If a guard is the only predecessor of its successors, the ordering and structure selection for structured control flow can be used. For guards that have successors with multiple predecessors, GOTO's are generated.

The key insight is that, although a node can be control dependent on many nodes, only one of these dependences may be from a structured construct. Observe that in a connected subpart of G_{cd}, when guards are created from GOTOs outside the partition into the subpart, the guards with the highest order numbers will be generated first. One or two GOTOs may result. When a GOTO will result in a guarded GOTO and a structured construct, care is taken to generate the GOTO first. In this case the node with larger relative number between the two guards will be selected, and a GOTO for it is generated.

The recursive generation of successors and their descendants must choose the lowest numbered successor to generate first. In structured code this is guaranteed to be the true branch, but with an IF-GOTO the false branch is lower. In structured code, the generation of successors is immediately preceded by their one and only predecessor. In unstructured code, to ensure all control dependences are satisfied, the recursion must cease if a node has other predecessors that have not yet been generated. When there are multiple GOTO's this situation may arise.

Returning to Example 4.2 and applying code generation results in the code below.

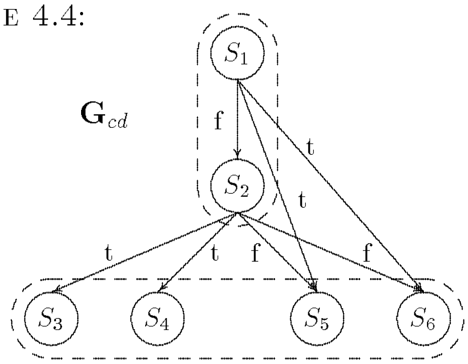
```
        DO I = 1, N
              EV1[I] = p1
S1            IF (EV1[I] .EQ. TRUE) GOTO 6
              S2
        6     S6
        ENDDO
        DO I = 1, N
              IF (EV1[I] .EQ. FALSE) GOTO 5
              IF (EV1 .EQ TRUE) THEN
S4                  IF (p4) GOTO P2
        5           S5
P2            CONTINUE
        ENDDO
```

Notice that when the second partition is generated the GOTO is generated first. The guards for S_4 and S_5 have the same order number, i.e. 1, but because S_1 was a GOTO, the jump to S_5 is generated first. Then S_4's guard, S_4, and S_5 are generated. Here and in Example 4.4 there are jumps into structured constructs. Although these jumps are non-standard Fortran, some compilers accept them and regardless can be implemented with GOTO's.

Finally, consider Example 4.4 with a distribution partition (S_1, S_2) and (S_3, S_4, S_5, S_6) where nodes are control dependent on more than one predecessor.

EXAMPLE 4.4:

```
        DO I = 1, N
S1            IF (p1) GOTO 5
S2            IF (p2) THEN
                    S3
                    S4
              ELSE
        5           S5
                    S6
              ENDIF
        ENDDO
```



Distribution restructuring, label renaming, and code generation performed on the above results in the following code.

```
        DO I = 1, N
              EV_2[I] = ⊤
S_1           EV_1[I] = p1
              IF (EV_1[I] .EQ. TRUE) GOTO P_1
S_2           EV_2[I] = p2
        P_1   CONTINUE
        ENDDO
        DO I = 1, N
              IF (EV_1[I] .EQ. TRUE) GOTO 5
              IF (EV_2[I] .EQ. TRUE) THEN
                    S_3
                    S_4
              ELSE IF (EV_2[I] .EQ. FALSE) THEN
        5           S_5
                    S_6
              ENDIF
        ENDDO
```

4.3 Other transformations

We now describe the extensions and algorithms for a selection of other loop transformations.

4.3.1 Loop skewing

Loop skewing is always safe, regardless of the type of control flow, because it does not change the order in which array memory locations are accessed. It only changes the shape of the iteration space (see Section 3.4.2).

4.3.2 Loop reversal

Loop reversal reverses the order of loop iterations. In the absence of control flow, it is safe if there are no loop-carried data dependences. This safety test is easily extended to handle control dependences. If the control dependences are loop-independent (*i.e.* they are completely internal to the loop nest), then they do not inhibit loop reversal. However, if they are loop-carried (*i.e.* exit branches) then loop reversal is not safe.

4.3.3 Loop permutation

Because loop permutations may be performed as a series of pair-wise interchanges, the rest of this discussion is simplified by focusing on loop interchange. Loop interchange is safe if it does not reverse the order of execution of the source and sink of any data dependence. By examining the direction vectors for all dependences carried on the outer loop, one may determine if there exists any data dependence with a direction vector of the form $(< >)$ which would be reversed by loop interchange. These data dependences are called *interchange-preventing*.

First consider nested loops containing internal branching. The tests for loop permutation are only concerned with preserving the original flow of values and although internal branching affects this flow, data dependence fully characterizes it. Therefore, the existing safety test suffices for this case.

An exit branch completely out of a nest inhibits permutation of any loops in the nest. Such permutation might result in executing too many iterations of some loops and too few of others. In a more restrictive setting where the exit branch is only out of a inner subset of the nest, permutation on the outer subset may be possible. A simple way to understand this effect is to think of it as a direction vector. Consider exit branches out of the k inner loops of a perfect loop nest of depth n to have a control dependence direction vector $(=_1 =_2 \ldots *_{n-k} \ldots *_{n-1} *_n)$. The direction vector indicates that loop levels $(n-k)$ to n may not be moved, but that levels 1 to $(n-k-1)$ are free to be permuted.

4.3.4 Strip mining

Because strip mining does not change the loop body or the iteration space, no additional mechanisms are needed when the loop body contains any type of control flow.

4.3.5 Privatization

Any variable can be made private to a loop if it is defined and used only within the loop and it is always defined before it is used. To determine if these conditions hold for scalar variables requires general data-flow information. When testing these conditions for arrays, data-flow and dependence information are required. General data-flow analysis is sophisticated enough to determine these conditions for scalars in loops containing control flow [CF87].

4.3.6 Scalar expansion

In vectorization, scalar expansion is often preferable to privatization. However, it is not clear in many cases which of the two is preferred. Scalar expansion has similar, but slightly less restrictive constraints than scalar privatization. Scalar expansion also requires that the variable be defined before it is used on all iterations of the loop, but it may still be live outside of the loop. Consider the following example of scalar expansion.

DO I = 1, N	*becomes*	DO I = 1, N
T = ...		TA(I) = ...
... = T		... = TA(I)
ENDDO		ENDDO
...		T = TA(N)
... = T		...
		... = T

Notice that the value stored on the last iteration of the loop, TA(N), must be stored into the scalar T . If there is no use of T outside the loop this is not necessary.[3] Again, data-flow analysis is able to determine these conditions in the loop and if the last value is always stored back, live analysis of T is unnecessary.

4.3.7 Loop fusion

Loop fusion places the bodies of two adjacent loops with the same number of iterations into a single loop [AC72]. Fusion is safe for two loops l_1 and l_2 if it does not result in values flowing from statements originally in l_2 back into statements originally in l_1 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.

If either loop contains internal branching, then the same test for safety is correct because the control dependences have the same effect in the fused code as in the original.

[3] Privatization could also utilize a store back, allowing it to be applicable even when loop definitions reach outside the loop.

Exit branches

Fusion is unsafe if there are exits out of the first loop which bypass execution of the second, so that the second loop header is control dependent upon the exit branches. The control dependences indicate that every exit test must execute before any of the statements in the second loop may be determined to execute.

If there are no exit branches out of the first loop and there is an exit branch in the second loop, it is possible to fuse correctly by restructuring the fused loops based on control dependence. The test for safety in this case need only determine if the data dependences prevent fusion. The restructuring step needs to replicate the control dependences for the second loop in the fused loop. Consider Example 4.5 and its corresponding control dependence graph.

Notice that the exit branches in the second loop result in a cyclic control dependence on the header. To maintain this control dependence structure for the statements in the second loop an execution variable EV is inserted and the code is slightly restructured.

A scalar execution variable EV records which exit branch, if any, is taken. Outside the loop, it is initialized to $\top$, which means that no exit branch has been taken. In the fused loop, the execution variable is assigned the label of the branch at the point an exit branch would be taken. The exit branch, GOTO *label*, is replaced with a GOTO to the end of the loop (10 CONTINUE in our example). The restructuring is completed by mimicking the original control dependence structure on the DO. An IF statement is inserted which dominates the execution of all the statements originating in the second nest. The test for the IF is false when an exit branch was taken on a previous iteration.

4.3.8 Loop peeling

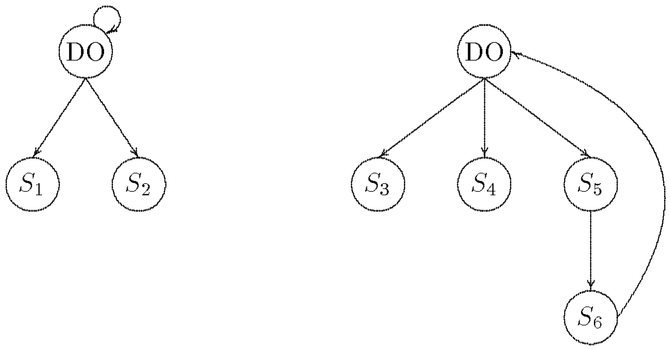
Peeling takes the statements in one or more iterations of a loop, executes them outside of the loop, and adjusts the loop bounds accordingly. Peeling may be performed on the first or last iteration. A slightly more general form of peeling, *index set splitting*, places a number of peeled iterations in a pre-loop or a post-loop. The considerations that arise from all of these are basically equivalent, so consider peeling the first iteration of a loop as seen below.

DO I = lb, ub, ss
    SL(I)
ENDDO
	becomes	$SL(lb)$
		DO I = $lb + ss$, ub, ss
		    SL(I)
		ENDDO

EXAMPLE 4.5: **Fusion**

```
     DO I = 1, N             becomes    EV = ⊤
          S1                            DO I = 1, N
          S2                                 S1
     ENDDO                                   S2
     DO I = 1, N                        IF (EV .EQ. ⊤) THEN
          S3                                 S3
          S4                                 S4
S5        IF (test5) GOTO ex1                IF (test5) THEN
S6        IF (test6) GOTO ex2                     EV = ex1
     ENDDO                                        GOTO 10
     SL8                                     ENDIF
     ...                                     IF (test6) THEN
ex1  SL9                                          EV = ex2
ex2  SL10                                         GOTO 10
                                             ENDIF
                                  10         CONTINUE
                                        ENDDO
                                        IF (EV .NE. ⊤ ) GOTO EV
                                        SL8
                                   ...
                                   ex   SL9
                                   ex   SL10
```

Control dependence graph



Peeling the first iteration replicates every statement in the loop, where every occurrence of the induction variable is replaced with the loop's lower bound. The new loop lower bound is the step size plus the original.

Peeling is always legal for loops with or without control flow because it does not change the order of statement execution. However, because peeling replicates statements, some care must be taken when labeled statements due to branching are present. For each peeled statement that is labeled, a new unused label must replace it. All references to that label in the peeled statements must also be changed to reflect the new label. Of course, the destinations of exit branches should remain as they were.

4.4 Related work

One approach taken in automatic vectorizers when loops contain conditional control flow is to convert control dependences into data dependences using a technique called *if-conversion* [All83, AKPW83]. If-conversion is theoretically appealing because it allows for a unified treatment of data dependence without control dependences, and has been used successfully in a number of vectorization systems [KKLW84, SK86]. However, it has several drawbacks.

In its original form, if-conversion is intractable and may introduce an exponential number of new logical arrays to record control flow decisions. Unfortunately, even when applied in more limited circumstances, it results in substantial increases in code space used for holding the results of conditionals. In addition, after if-conversion has been performed, it is not easy to reconstruct the original program, or even efficient branching code, if vectorization fails. Another concern is that the transformed code no longer resembles the original. Even though many other transformation have this problem to some degree, if-conversion typically obliterates the original program. These drawbacks are exacerbated in an interactive environment and are significant enough that other solutions have been sought.

An alternative approach when control flow is present uses explicit control and data dependences. In his dissertation, Towle develops techniques for vectorizing programs with control flow using loop distribution, scalar expansion, and replication using data and control dependence information [Tow76]. However, his definition of control dependence embeds, but does not extract the essential control relationship between two statements, that is if the execution of one statement directly determines the execution of another.

Control dependence as formulated by Ferrante, Ottenstein, and Warren is clean, complete, and extracts this essential relationship [FOW87]. They include control and data dependences in the program dependence graph, PDG, and our approach uses the same basis. Their paper also discusses several optimizing transformations performed on the PDG: node splitting, code motion, loop fusion, and loop peeling. However, their algorithms are applicable only for structured control flow. Neither Towle or Ferrante *et al.* present loop distribution.

Ferrante, Mace, and Simons present related algorithms whose goals are to avoid replication and branch variables when possible [FM85, FMS88]. Their code generation algorithms convert parallel programs into sequential ones, and like ours, are based on G_{cd}. They discuss three transformations that restructure control flow: loop fusion, dead code elimination, and branch deletion.

Callahan and Kalem present two methods for generating loop distributions in the presence of control flow [CK87a]. The first, which works for structured or unstructured control flow, replicates the control flow of the original loop in each of the new loops by using G_f. *Branch variables* are inserted to record decisions made in one loop and used in other loops. An additional pass then trims the new loops of any empty control flow. Dietz uses a very similar approach [Die88]. These approaches have some of the same drawbacks of if-conversion, *i.e.* the proliferation of unnecessary guards.

Callahan and Kalem's second method, which works only for structured control flow, uses G_f, G_{cd}, and Boolean execution variables. Their execution variables indicate if a particular node in G_f is reached and they are created for edges in G_{cd} that cross between partitions. Their execution variables are assigned true at the successor indicating the successor will execute, rather than assigning the decision made at the predecessor. Also, one execution variable may be needed for every successor in the descendant partition. Because their code generation algorithm is based on G_f, rather than G_{cd}, the proof of how an execution variable is used is much more difficult and is not given. Towle [Tow76] and Baxter and Bauer [BB89] use similar approaches for inserting conditional arrays.

The Stardent compiler distributes loops with structured control flow by keeping groups of statements with the same control flow constraints together [All90]. For example, all the statements in the true branch of a block IF must stay together, so only the outer level of IF nests can be considered. This limits effectiveness of distribution because partitions are artificially made larger, possibly by grouping parallel statements with sequential ones.

4.5 Discussion

In summary, although much attention has been paid to modeling and understanding control flow in other work, a general formulation of parallelism enhancing transformations with arbitrary control flow was not available until now. Using control and data dependences, we have presented new and generalized versions of many important loop transformations. In particular, the algorithm for loop distribution was shown optimal and represents a significant improvement over previous algorithms.

I myself have never been able to find out precisely what feminism is; I only know that people call me a feminist whenever I express sentiments that differentiate me from a doormat. Rebecca West, 1913.

Chapter 5

Interprocedural Transformations

Striving for a large granularity of parallelism has a natural consequence; the compiler must look for parallelism in regions of the program that span multiple procedures. This kind of optimization is called *whole program* or *interprocedural* analysis and transformation. This chapter presents a new approach that enables compiler optimization of procedure calls and loop nests containing procedure calls. We introduce two interprocedural transformations, loop extraction and loop embedding, that move loops across procedure boundaries, exposing them to loop nest optimizations. We also describe the efficient support of these transformations using the interprocedural compilation system in the ParaScope parallel programming environment. These transformations are shown effective in practice on existing applications programs.

5.1 Introduction

To expose parallelism and computation for parallel 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. Good software engineering practices encourage modularity as a way to manage program computation and complexity, and increasingly, programmers are using a modular programming style. Therefore, it is natural to expect that programs will contain many procedure calls and procedure calls in loops, and to ensure high performance compilers will need to optimize them.

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

1. **Interprocedural analysis** applies data-flow analysis techniques across procedure boundaries to enhance the effectiveness of dependence testing. Regular section analysis is a sophisticated form of interprocedural analysis which makes

it possible to parallelize loops with calls (see Section 2.3). It determines if the side effects to arrays as a result of each call are limited to nonintersecting subarrays on different loop iterations [CK87b, HK90].

2. **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 [AC72].

Even though regular section analysis and inlining are frequently successful at enabling optimization, each of these methods has its limitations [HK90, LY88a, Hus82]. 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 increase in code size which may disastrously increase compile time and seriously inhibit separate compilation [CHT91, RG89]. 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* [BCHT90].

In this chapter, 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. However, because there is a cost for interprocedural transformations, our strategy applies them only when performance benefits are expected to result.

The performance benefit of these transformations comes from using the exposed loops in high-payoff *intraprocedural* loop optimizations. Any intraprocedural transformations that requires loop nests may be applicable on those provided by loop embedding and extraction. Additionally, testing the safety and profitability of some of the loop transformations across procedure boundaries requires no extension to the tests discussed in Chapters 3 and 4. Extensions are needed for transformations that require dependence distance information such as loop permutation. The intraprocedural optimizations which are extended in this chapter are loop fusion and loop permutation. These results easily generalize for other transformations such as loop skewing [Wol86] and unroll and jam [CCK88].

As a motivating example, consider the Fortran code in Example 5.1(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 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

---

EXAMPLE 5.1: **Loop embedding**

```
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
PARALLEL DO J = 1, 20
    F(J,I) = F(J,I-1) + 9
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
    PARALLEL DO J = 1, 20
       F(J,I) = F(J,I-1) + 9
    ENDDO
ENDDO
```

(b) loop embedding

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


CALL S(A)

SUBROUTINE S(F)
REAL F(N,N)
INTEGER I,J
PARALLEL DO J = 1, 20
    DO I = 1, 100
       F(J,I) = F(J,I-1) + 9
    ENDDO
ENDPARDO
```

(c) loop interchange

---

to cases where it is expected to be profitable. This strategy, called *goal-directed interprocedural optimization*, avoids the costs of interprocedural optimization when it does not enable other performance enhancing optimizations [BCHT90]. Interprocedural transformations are applied as dictated by a code generation algorithm that explores possible transformations, selecting a choice that introduces parallelism and exploits data locality.

The code generator is part of an interprocedural compilation system that efficiently supports interprocedural analysis and optimization by retaining separate compilation of procedures. We first explored this type of system using a simple, performance estimation based parallel code generation algorithm [HKT91]. This chapter provides a more general framework and that is integrated into a more sophisticated parallelization algorithm discussed in Chapter 7. We also present experimental results to illustrate the efficacy of these transformations on application programs.

5.2 Technical background

5.2.1 Augmented call graph

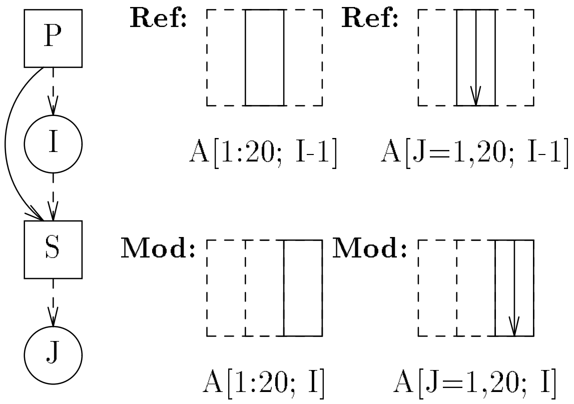
The program representation for interprocedural transformations requires an *augmented call graph* (G_{ac}) which describes the calling relationships among procedures and loop nests. The details of the G_{ac} are presented in Section 2.3. Figure 5.1(a) shows an abbreviated version of the augmented call graph G_{ac} for the program from Example 5.1, where the solid line is a call edge and the dashed lines are nesting edges.

5.2.2 Interprocedural 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 efficient [HK90]. 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.

FIGURE 5.1: **Sections and data access descriptors**



(a) G_{ac} (b) Sections (c) DAD

Sections are separated into modified and referenced sets. The sections for Example 5.1 are shown in Figure 5.1(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.

Regular sections enable dependence analysis to determine if loops containing calls are parallel and are sufficient to determine the safety of intraprocedural transformations on a loop nest containing calls. However, a more precise version of sections is needed to determine the safety of intraprocedural transformations which involve loops in different procedures before loop embedding or extraction places them in the same procedure. These are similar to *data access descriptors* or DADs and they provide detailed information about references and how the loops in a called procedure access it [BK89]. Our version of DADs are a little more precise because we have the loop header information in G_{ac}, but for the purposes of this discussion DADs evoke the appropriate meaning.

A DAD identifies the section of an array accessed and the order of that access in terms of each enclosing loop's index expression. It also indicates the relative ordering of the accesses. We consider DADs as annotations of sections. In addition, the sections are marked as exact or inexact for the purposes of dependence testing used in determining the safety of intraprocedural transformations in the caller. The regular section information is sufficient to test dependence on loops containing calls. To test dependence on the loops in the call at the call site demands that the DAD be exact in the following sense. An exact reference or modified section must be be described in terms of a constant or any surrounding loops. It must also meet one of the following criteria either (1) it is not the result of a merge, or (2) if it is the result of a merge, either the merge was between accesses where they overlap exactly and completely or the accesses are completely disjoint. Figure 5.1(c) illustrates the DAD annotations for the program in Example 5.1.

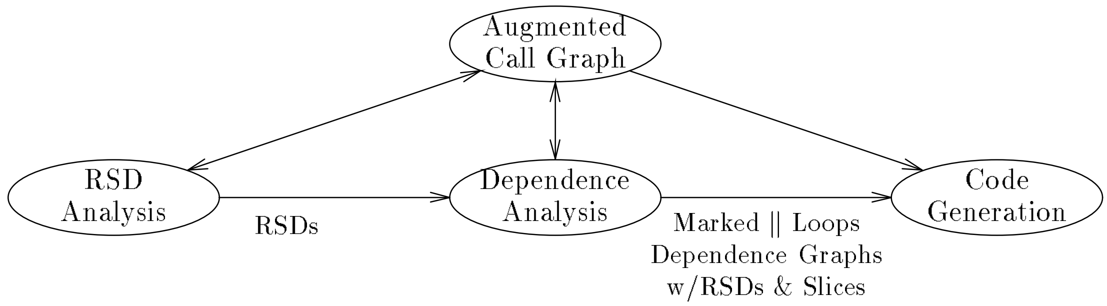
5.3 Support for interprocedural optimization

In this section, we present the compilation system of the ParaScope Programming Environment [CCH+88, CKT86a]. 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 [CKT86b]. This system was original intended for scalar compilation. This section extend the ParaScope system to support parallel code generation.

5.3.1 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 of global variables and call-by-reference formal parameters that are locally modified or referenced in the procedure. The procedure's calling interface is also determined in this phase. It includes descrip-

FIGURE 5.2: **Information flow for interprocedural transformations**



tions 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 [CKT86a, Hal91]. The program compiler first builds the augmented call graph described in Section 2.3. 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 yields dependence edges that are placed in the dependence graph. If the source or sink of a dependence is a call site, a section 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 its source or dependence graph.

Planning and code generation

The final phase of the program compiler determines where interprocedural optimization is estimated to be profitable. Planning is important to interprocedural optimization since unnecessary transformations may lead to significant compile-time costs without any execution-time benefit. To determine the safety of transformations, the dependence graph and sections are sufficient.

The relationship among the compilation phases is depicted in Figure 5.2. 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.

Procedure cloning

Procedures optimized with loop embedding or extraction may have multiple callers, and an optimization valid for one caller may not be valid for another. To avoid 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 [CKT86a, CHK92].

5.3.2 Recompilation analysis

A unique part of the ParaScope compilation system is its recompilation analysis, which avoids unnecessary recompilation after program edits. Recompilation analysis tests that interprocedural facts used to optimize a procedure have not been invalidated by editing changes [CKT86b, BC86, BCKT90]. 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. As a result, 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 [Hal91].

5.4 Interprocedural transformation

Loop extraction and loop embedding expose the loop structure to optimization without incurring the costs of inlining. Just as inlining is always safe, these transformations are always safe. The mechanics of performing the movement of a loop header is detailed below. If moving additional statements is desired, it may be performed with the techniques developed for inlining.

5.4.1 Loop extraction

Loop extraction moves a loop that encloses the body of its 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 5.2(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`. The `J` loops may now be fused and interchanged to improve performance, as in Example 5.2(c).

5.4.2 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

Example 5.2:

(a) before transformation	(b) loop extraction	(c) loop fusion & interchange

<pre>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</pre>
	<pre>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</pre>
		<pre>SUBROUTINE P(A)
		REAL A(N,N), B(N,N)
		INTEGER I,J
		
		DO J = 1, 100
		  DO I = 1, 3
		    CALL S(A,I,J)
		    CALL S(B,I,J)
		  ENDDO
		ENDDO</pre>

<pre>SUBROUTINE S(F,I)
REAL F(N,N)
INTEGER I,J
DO J = 1,100
  F(J,I) = F(J,I) + 9
ENDDO</pre>
	<pre>SUBROUTINE S(F,I,J)
	REAL F(N,N)
	INTEGER I,J
	
	F(J,I) = F(J,I) + 9</pre>
		<pre>SUBROUTINE S(F,I,J)
		REAL F(N,N)
		INTEGER I,J
		
		F(J,I) = F(J,I) + 9</pre>

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 5.1.

If the index variable of the loop to be embedded appears in an actual parameter at the call site, this parameter is no longer correctly defined. To remedy this problem, the formal parameters in the call 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 [BC86]. 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 our implementation of inlining.

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 may not be necessary. However, if further optimization is desired, updating the dependence information is straightforward. The dependence information just moves and translates with the loop which is moving.

Embedding versus Extraction

There are several factors which affect the choice between embedding or extraction during the optimization process. All things being equal, embedding loops needed for optimizations into the called procedure is preferable because it reduces procedure call overhead. However, if several loops originating from different call sites are needed to perform an optimization, extraction is required (as illustrated in Example 5.2). If an optimization uses a loop in the call and more than one loop of the caller, then loop extraction is also preferred. On the other hand, if the optimization involves the inner loop of the caller and more than one loop in the called procedure, loop embedding is preferred. The other option for these and other more complex circumstances is to perform loop embedding or extraction multiple times to adjoin the necessary loops.

5.5 Intraprocedural transformations

The following two sections discuss how to test for the safety of intraprocedural transformations across procedure boundaries. The tests are needed when the requisite loops are not in the same procedure, but may be placed together via embedding or extraction.

5.5.1 Loop fusion

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). Once loops are exposed, fusion and other optimizations may be performed as illustrated by Example 5.2. In the algorithm *checkFusion*, we consider fusion of $\{s_1, s_2\}$, where s_i

is either a call or a loop. Loop fusion is restricted in this setting in that there may not be any intervening statements between s_1 and s_2.

The test for fusion between two loops, l_1 and l_2, 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 section and its DAD correspond to the loops being considered for fusion and are tested identically to variable references (see Section 5.2.2). Unfortunately, while variable references are always exact, a section is not. If a section for a particular array is not exact and a potential dependence exists between the loop nests, fusion is conservatively assumed to be unsafe. (There exists a potential dependence if an array is referenced in both nests and at least one is a write.) A more precise test could be performed by inspecting the dependence graphs for each called procedure. In practice, the more precise test may be no more successful and could introduce significant overhead.

5.5.2 Loop permutation

Loop permutation of a loop nest rearranges the loop headers, changing the order in which the iteration space is traversed. As with fusion, the distance/direction vectors for the loops in the caller being considered in the permutation must be computable

---

ALGORITHM 5.1: **Interprocedural fusion test**

checkFusion (s_1, s_2)
INPUT: (s_1, s_2), where s_i is a call or a loop and s_1 is adjacent to s_2
OUTPUT: returns true if fusion is safe
ALGORITHM:
let l_1 = the loop header of s_1
let l_2 = the loop header of s_2
if the number of iterations of l_1 differ from l_2
return false
for each forward dependence $(src_{s_1}, sink_{s_2})$
if src_{s_1} or $sink_{s_2}$ is not exact
return false
if $(src_{s_1}, sink_{s_2})$ becomes backward loop-carried
return false
endfor
return true

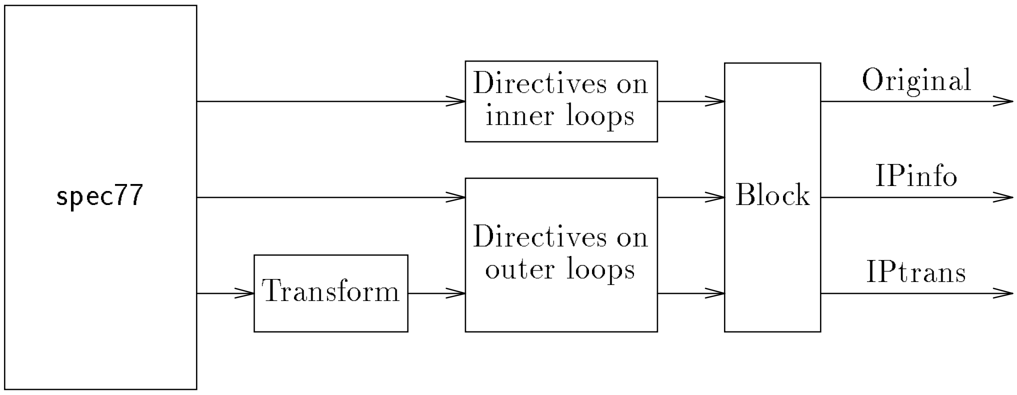
---

at the call. Again, the sections involved in dependences must be exact or the test conservatively assumes the transformation to be unsafe. Conversely, in the call the distance/direction vectors for surrounding loops could be made available when the call is being optimized. This option is less appealing because other optimizations are inhibited in the caller. Regardless, if there are loops in either the caller or the called routine that do not carry any dependences, the augmented call graph reflects it and many permutations can be shown safe without additional dependence testing.

5.6 Experimental results

This section presents significant performance improvements due to interprocedural transformation on two scientific programs, *spec77* and *ocean*, taken from the Perfect Benchmarks [CKPK90]. To locate opportunities for transformations, we browsed the dependences in the program using the ParaScope Editor [BKK+89, KMT91a, KMT91b]. 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 entire application was not parallelized. The original and optimized programs were executed on a 20-processor Sequent Symmetry S81. Since the optimizations used differed slightly for each program, they are described separately.

FIGURE 5.3: **Stages of preparing program versions for experiment**



5.6.1 Spec77

Spec77 contains 3278 non-comment lines and is a fluid dynamics weather simulation that uses Fast Fourier Transforms and rapid elliptic problem solvers. 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 5.3 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 strip mined to allow multiple consecutive iterations to execute on the same processor without synchronization. The compiler default is to schedule each iteration of a parallel loop separately, incurring additional overhead.

	processors = 7		processors = 19	
	time in optimized portion	speed-up	time in optimized portion	speed-up
Original	81.9s	5.7	45.8s	10.1
IPinfo	80.0s	5.8	48.0s	9.7
IPtrans	80.6s	5.8	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 portion 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.

5.6.2 Ocean

Ocean has 1902 non-comment lines and is a 2-D fluid dynamics ocean simulation that uses Fast Fourier Transforms. 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 5.3. 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	speed-up
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.

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.

5.7 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 [AJ90, CHT91, Hus82] and interprocedural analysis of array side effects [BK89, BC86, CK87b, HK90, HHL90a, HHL90b, LY88a, LY88b, TIF86]. 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 selection of 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 [HK90]. 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 [LY88a, LY88b, TIF86].

5.8 Discussion

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 evident 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. Increasing numbers of scientific programmers are using a modular programming style and cannot afford to pay a performance penalty. By providing compiler support to efficiently 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. These techniques enable a desirable programming style which uses procedures that can be effectively parallelized.

This chapter and the previous one provide algorithmic support for applying transformations to entire applications. In particular, program optimization is enabled for loops containing control flow and is not inhibited when loop nests span procedure boundaries. We now turn to the proper application of these transformations to effect excellent parallel performance.

Chapter 6

Optimizing for Parallelism and Data Locality

Previous research has used program transformation to introduce parallelism and to exploit data locality. Unfortunately, these two objectives have usually been considered independently. This chapter explores the tradeoffs between effectively utilizing parallelism and memory hierarchy on shared-memory multiprocessors. We present a simple, but surprisingly accurate, memory model to determine cache line reuse from both multiple accesses to the same memory location and from consecutive memory access. The model is used in memory optimizing and loop parallelization algorithms that effectively exploit data locality and parallelism in concert. We demonstrate the efficacy of this approach with very encouraging experimental results. This algorithm forms the core of our parallel code generation strategy.

6.1 Introduction

Transformations to exploit parallelism and to improve data locality are two of the most valuable compiler techniques in use today. Independently, each of these optimizations has been shown to result in dramatic improvements. This chapter seeks to combine the benefits of both by using a simple memory model to drive optimizations for data locality and parallelism. By unifying the treatment of these optimizations, we are able to place loops with data reuse on inner loops and to introduce parallelism for outer loops. Our strategy produces data locality at the innermost loops, where it is most likely to be exploited by the hardware and places parallelism at the outermost loop, where it is most effective. If these two goals conflict, we present an algorithm that usually reaps the benefits of both.

Optimizing data locality is necessarily both architecture and language dependent. However, the reuse of memory locations and the consecutive access of adjacent memory locations form the foundation of most memory hierarchy optimizations. Reuse of a particular memory reference for arrays can be discovered using data-dependence analysis. However, reuse of consecutive accesses, often called *unit stride* access, is a significant source of reuse that can easily be determined when the storage order of arrays and the cache line size is known. In this chapter we introduce a simple model for estimating the cost, in memory references, of executing a given loop nest. The principal advantage of this model over previous models is that it takes into account cache reuse due to consecutive accesses to the same cache line. We show how this model can be used to exploit data locality at multiple levels via loop permutation.

Our algorithm first uses the memory model to find a loop organization that exploits data locality. It then seeks to parallelize the outermost loop or a parallel loop that can be positioned outermost. Given sufficient iterations, it then strip mines the loop into two loops, such that one loop is used to achieve locality and the other is used to introduce parallelism.

Matrix multiply example

As an example of this process, consider the ubiquitous matrix multiply.

```
DO J = 1, N
    DO K = 1, N
        DO I = 1, N
            C(I,J) = C(I,J) + A(I,K) * B(K,J)
```

Assuming arrays are stored such that columns of the arrays are in consecutive memory locations, *i.e. column-major order*, this loop organization exploits data locality in the following manner. The consecutive access on the inner I loop to C(I,J) and A(I,K) provide an opportunity for cache line reuse when the cache line size is greater than 1. There is also a loop-invariant reuse of B(K,J) on the I loop. Additionally, the J and the I loops can be parallel. However, if the number of processors, P, is less than the number of iterations of either loop, it is not profitable to utilize both levels of parallelism at once due to additional scheduling overhead. A better execution time would result by maximizing the granularity of one level of the parallelism and then matching it to the machine. If N = P, selecting J to be executed in parallel preserves data locality and introduces a single level of parallelism with maximum granularity.

```
PARALLEL DO J = 1, N
    DO K = 1, N
        DO I = 1, N
            C(I,J) = C(I,J) + A(I,K) * B(K,J)
```

However, if the number of loop iterations is greater than the number of processors, N > P, it is often useful to combine independent iterations into a single parallel task to achieve granularity that matches the machine. The parallel loop is strip mined by the number of processors where the strip size is SS = $\lceil$ N/P $\rceil$. We call the J loop the strip and the JJ loop, which walks between strips, the iterator.

```
PARALLEL DO JJ = 1, N, SS
      DO J = JJ, MIN(JJ + SS - 1, N)
            DO K = 1, N
                  DO I = I, N
                        C(I,J) = C(I,J) + A(I,K) * B(K,J)
```

The parallel JJ loop carves up the data space nicely, but if each processor's cache is still not large enough to contain all of array A, *tiling* the loop nest further improves performance by providing reuse of A. Tiling combines strip mining and loop interchange to promote reuse across a loop nest [IT88, Wol89a]. For matrix multiply, the loop nest may be tiled by strip mining the K loop by TS and then interchanging it with J.

```
PARALLEL DO JJ = 1, N, SS
      DO KK = 1, N, TS
            DO J = JJ, MIN(JJ + SS - 1, N)
                  DO K = KK, MIN(KK + B - 1, N)
                        DO I = I, N
                           C(I,J) = C(I,J) + A(I,K) * B(K,J)
```

Here, TS is selected based on the cache size. This organization moves the reuse of A(1:N,KK:KK+TS-1) on the J loop closer together in time, making it more likely to still be in cache. This optimization approach may be divided into three phases:

1. optimizing to improve data locality,
2. finding and positioning a parallel loop, and
3. performing low-level memory optimizations such as tiling for cache and placing references in registers [LRW91, CCK90].

This chapter focuses on the first two phases. We advocate the first two phases be followed by a low-level memory optimizing phase, but do not address it here.

6.2 Memory and language model

Because we are evaluating reuse, we require some knowledge of the memory hierarchy. However, because our model is very simple, only minimal knowledge of the cache is required; the compiler must know the cache line size (cls). The size, set associativity, and replacement policy of the cache are not important here. In addition, we assume a write-back cache and ignore non-unique write references. If the cache is write-through, these writes should be included.

In addition, we only concern ourselves with memory accesses caused by array references, since they dominate memory access in scientific Fortran codes. We also assume that arrays are stored in *column-major* order, where unit stride accesses in the first array dimension translate into contiguous memory accesses. Our results are also valid for *row-major* arrays such as those found in C with only minor changes.

6.3 Tradeoffs in optimization

This section illustrates with an experiment the influence of memory reuse and parallelism granularity on speed-up. As expected, it indicates the best performance is possible only when both are utilized effectively in concert. It also shows that when both cannot be achieved at once, there are situations where favoring one or the other results in the best execution time. Neither always dominates. To illustrate, we phrase the following question.

> *Given enough computation to make parallelism profitable, what is the effect of reuse and how should it affect the optimization strategy?*

Figure 6.1 presents the results of executing different parallel versions of the following loop nest on 18 processors of a Sequent Symmetry S81 with 20 processors, with increasing amounts of total work.

```
DO J = 1, N
    DO I = 1, M
        DO H = 1, L
            C(I, J) = C(I, J) + A(I, J) + B(I, J)
```

The total amount of work is increased by varying the upper bounds N and M from 2 to the number of processors (P = 18). We consider positioning I or J as the outer parallel loop in the nest. In Figure 6.1, the *best* version of this loop nest has an outer parallel J loop with 18 iterations (N = 18) and total work is increased by varying M from 2 to 18. Each of the 18 processors accesses distinct columns of each array. This organization exploits cache line reuse on each processor and results in linearly-scalable speed-up.

When the J loop is outermost and the number of parallel iterations of is varied from 2 to 18 along with P and the I loop contains 18 iterations, the total amount of

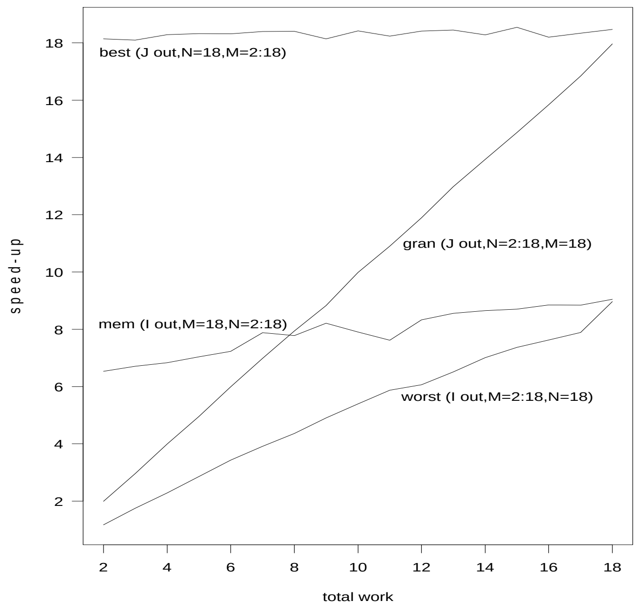


FIGURE 6.1: **Memory and parallelism tradeoffs**

work increases, but the work per processor remains the same. This organization is illustrated by the *gran* line. In this case, the speed-up scales by the number of parallel iterations, but cache line reuse is still facilitated on each processor.

If instead the I loop is made outermost and parallel, then processors must compete for the cache line which contains C(I,J) in order to write it. This competition is called *false sharing*. In addition, multiple processors require cache lines containing A(I,J) and B(I,J), increasing network contention and total memory utilization. When the number of parallel iterations of the I loop as outermost varies from 2 to 18 along with P and the J loop contains 18 iterations, the *worst* line indicates the performance. If the number of parallel iterations of I is held at 18 while the J loop is varied from 2 to 18, the *mem* line results.

Compare the pair of lines *best* and *mem*. The factor of two difference is due to the benefit of cache line reuse in *best*, and the limitations of false sharing and increased bus and memory utilization in *mem*. The same comparison holds for the *gran* and *worst* lines. These results indicate that the parallelizing algorithm must recognize reuse and false sharing to be effective.

Now compare the pair of crossing lines *gran* and *mem*. These computations differ only by an interchange. An optimization strategy that only used loop interchange would be forced to pick between the two. To obtain the best performance for this example, the J loop would be outermost when N > 8, otherwise the I loop should be outermost. In addition, this "crossover" point would need to be determined for each computation, a daunting task. Our approach instead combines loop interchange and strip mining in a parallelization strategy that minimizes false sharing and exploits data reuse.

6.4 Optimizing data locality

In this section we describe two sources of data reuse, then we incorporate both in a simple yet realistic cost model. In subsequent sections, this cost model is used to guide optimizations for improving data locality and exploiting parallelism.

6.4.1 Sources of data reuse

We first consider the two major sources of data reuse.

- multiple accesses to the same memory location
- accesses to consecutive memory locations (*i.e.* stride 1 or unit stride access)

Multiple accesses to the same memory location may arise from either a single array reference or multiple array references. These accesses are loop-independent if they occur in the same loop iteration, and are loop-carried if they occur on different loop iterations. Wolf and Lam call this *temporal* reuse [WL91]. The most obvious source of temporal reuse is from loop-invariant references. For instance, consider the reference to A(J) in the following loop nest. It is invariant with respect to the I loop, and is reused by each iteration.

```
DO J = 1,N
    DO I = 1,N
        S = S + A(J) + B(I) + C(J,I)
```

A second source of data reuse is caused by multiple accesses to consecutive memory locations. For instance, each cache line is reused multiple times on the inner I loop for B(I) in the above example. Wolf and Lam call this *spatial* reuse [WL91]. The actual amount of reuse is dependent on the size of B(I) relative to the cache line size and the pattern of intervening references. For the rest of this chapter, we assume for simplicity that the cache line size is expressed as a multiple of the number of array elements. For reasonably large computations, references such as C(J,I) do not provide any reuse on the I loop, because the desired cache lines have been flushed by intervening memory accesses.

Previous researchers have studied techniques for improving locality of accesses for registers, cache, and pages [AS79, CCK90, WL91, GJG88]. We concentrate on improving the locality of accesses for cache; *i.e.* we attempt to increases the locality of access to the same cache line. Empirical results show that improving spatial reuse can be significantly more effective than techniques that consider temporal reuse alone [KMT92]. In addition, consecutive memory access results in reuse at all levels of the memory hierarchy except for registers.

6.4.2 Simplifying assumptions

To simplify analysis we make two assumptions. First, our loop cost function assumes that reuse occurs only across iterations of the innermost loop. This assumption decreases precision but greatly simplifies analysis, since it allows the number of cache line accesses to be calculated independent of the permutation of all outer loops. This assumption is accurate if the inner loop contains a sufficiently large number of memory accesses to completely flush the cache after executing all of its iterations. We show later that our optimizations to improve locality can select a desirable permutation of outer loops even with this restriction.

Cache interference refers to the situation where two memory locations are mapped to the same cache line, eliminating an opportunity to exploit reuse for one of the references. Our second assumption is that cache interferences occur rarely for small numbers of inner loop iterations, compared to the total number of distinct cache lines accessed in those iterations. In other words, we expect very few interferences for each cache line being reused, since the cache line is only needed for a small number of consecutive inner loop iterations. Lam *et al.* show that this assumption may not hold if cache lines must remain live for longer periods of time. Considerable interference may take place when loops are tiled to increase reuse across outer loops [LRW91].

6.4.3 Loop cost

Given these assumptions, we present a loop cost function *LoopCost* based on our memory model. Its goal is to estimate the total number of cache lines accessed when a candidate loop l is positioned as the innermost loop. The result is used to guide loop permutation to improve data locality. The estimate is computed in two steps. First, references that will access the same cache line in the same or different iterations of the l loop are combined using *RefGroup*. Second, the number of cache lines accessed by all groups is calculated using *LoopCost*.

6.4.4 Reference groups

The goal of the *RefGroup* algorithm is to partition variable references in the program text into *reference groups* such that all references in a group access the same memory locations, and consequently the same cache line. Wolf and Lam call these groups *equivalence classes* exhibiting *group-temporal* reuse. The partition process is particularly simple here because we only consider reuse for each loop when it is positioned innermost.

Two references are in the same reference group for loop l if they actually access some common memory location (data dependence $\vec{\delta}$ exists between them), and the

---

ALGORITHM 6.1: **Determine reference groups**

RefGroup (*Refs*, $\mathcal{DG}$, l)
INPUT:
$Refs = \{Ref_1 \ldots Ref_n\}$ references
$\mathcal{DG} = \{\langle Ref_i\ \vec{\delta}\ Ref_j\rangle, \ldots\}$ the dependence graph
l = candidate innermost loop

OUTPUT:
$\{RefGroup_1 \ldots RefGroup_m\}$ reference groups for l

ALGORITHM:
$m = 0$
while $Refs \neq \emptyset$ **do**
$m = m + 1$
$RefGroup_m = \{r\}$, where $r \in Refs$
$Refs = Refs - \{r\}$
for each $\langle r\ \vec{\delta}\ r'\rangle$ **or** $\langle r'\ \vec{\delta}\ r\rangle \in \mathcal{DG}$ *s.t.* $r' \in Refs$
if (δ_l is a constant d) & (δ_l is the only nonzero entry in $\vec{\delta}$)
$RefGroup_m = RefGroup_m + \{r'\}$
$Refs = Refs - \{r'\}$
endif
endfor
endwhile

---

reuse occurs on l if it is positioned as the innermost loop. The common accesses then occur on either the same iteration of l ($\delta_l = 0$) or across d iterations of l ($\delta_l = d$). More formally we define *RefGroup* as follows.

Definition 6.1 Two references Ref_1 and Ref_2 belong to the same *reference group* with respect to loop l if and only if:

1. $\exists\ Ref_1 \vec{\delta} Ref_2$, and
2. $\vec{\delta}$ is a loop-independent dependence, or
   δ_l, the entry in $\vec{\delta}$ corresponding to loop l, is a constant d (d may be zero) and all other entries are zero.

Jacobi example

For instance, consider the following Jacobi iteration example.

```
DO I = 2,N-1
     DO J = 2,N-1
          A(J,I) = 0.2* (B(J,I) + B(J-1,I) + B(J,I-1)
                    + B(J+1,I) + B(J,I+1))
```

Data dependences connect all references to B. The reference groups for the I loop are:

{A(J,I)}, {B(J,I),B(J,I-1),B(J,I+1)},
{B(J-1,I)}, {B(J+1,I)}.

The reference groups for the J loop are:

{A(J,I)}, {B(J,I),B(J-1,I),B(J+1,I)},
{B(J,I-1)}, {B(J,I+1)}.

RefGroup is shown in Algorithm 6.1. Its efficiency may be improved by pruning all identical array references, since they access the same memory location on each iteration and always fall in the same reference group.

6.4.5 Loop cost algorithm

After the number of reference groups for loop l is computed with *RefGroup*, the algorithm *RefCost* is applied to estimate the total number of cache lines that would accessed by each reference group if l were the innermost loop. Once again, the task is simplified because we only consider reuse between iterations of l.

RefCost works by considering one array reference *Ref* from each reference group; these representative references are classified as loop-invariant, consecutive, or non-consecutive with respect to loop l. Loop-invariant array references have subscripts that do not vary with l; they require only one cache line for all iterations of l.[4] Consecutive array accesses vary with l only in the first subscript dimension. They access a new cache line every cls iterations, resulting in $trip/cls$ cache line accesses, assuming l performs $trip$ iterations. Fewer cache lines are reused for nonunit strides. Non-consecutive array accesses vary with l in some other subscript dimension; they access a different cache line each iteration, yielding a total of $trip$ cache line accesses.

Once *RefCost* is computed, the algorithm *LoopCost* calculates the total number of cache lines accessed by all references when l is the innermost loop. It simply sums *RefCost* for all reference groups, then multiplies the result by the trip counts of all the remaining loops. This calculation will underestimate the number of cache lines accessed on the inner loop, if the distance of the dependences for a particular *RefGroup* set are greater than cls. Also, slight underestimates occurs because the exact alignment of arrays in memory is not known until run-time. *LoopCost* will overestimate the number of cache lines, if there is additional reuse across an outer loop.

LoopCost is expressed more formally in Algorithm 6.2 for the following loop nest containing one array reference from each reference group $RefGroup_1 \ldots RefGroup_m$:

do $i_1 = lb_1, ub_1, s_1$
  do $i_2 = lb_2, ub_2, s_2$
    ...
    do $i_n = lb_n, ub_n, s_n$
      $Ref_1(f_1(i_1, \ldots, i_n), \ldots, f_j(i_1, \ldots, i_n))$
      ...
      $Ref_m(g_1(i_1, \ldots, i_n), \ldots, g_k(i_1, \ldots, i_n))$

Note that *LoopCost* can be used to calculate cache line accesses even for array references with complex subscript expressions. For instance, it determines that A(I+J+N) results in consecutive memory accesses with respect to both the I and J loops.

[4]Of course, loop-invariant references should eventually be put in registers.

ALGORITHM 6.2: **Determine inner loop cost**

LoopCost $(\mathcal{L}, \mathcal{R}, cls)$

INPUT:

$$\begin{aligned}
\mathcal{L} &= \{l_1, \ldots, l_n\} \text{ a loop nest with headers } lb, ub, s \\
\mathcal{R} &= \{Ref_1, \ldots, Ref_m\} \text{ representatives from each reference group} \\
trip_l &= (ub_l - lb_l + s_l)/s_l \\
cls &= \text{the cache line size,} \\
appear(f) &= \text{the set of index variables that appears in} \\
&\quad\ \text{the subscript expression } f \\
coeff(i_l, f) &= \text{the coefficient of the index variable } i_l \text{ in the subscript } f \\
&\quad\ \text{(it may be zero)}
\end{aligned}$$

OUTPUT:

$$LoopCost(l) = \text{number of cache lines accessed with } l \text{ as innermost loop}$$

ALGORITHM:

$$\textbf{LoopCost}(l) \ = \ \sum_{k=1}^{m} \left(\textbf{RefCost}(Ref_k(f_1(i_1, \ldots, i_n), \ldots, f_j(i_1, \ldots, i_n))) \ * \prod_{h \neq l} trip_h \right)$$

$\textbf{RefCost}(Ref_k) \ =$

1	**if**	$(i_l \notin appear(f_1)) \wedge \ldots \wedge (i_l \notin appear(f_j))$	**loop invariant**
$trip_l/cls$	**if**	$(i_l \in appear(f_1)) \ \wedge (\lvert coeff(i_l, f_1)\rvert = 1) \wedge (\lvert s_l\rvert = 1) \ \wedge$ $(i_l \notin appear(f_2)) \wedge \ldots \wedge (i_l \notin appear(f_j))$	**unit stride**
$trip_l$	**otherwise**		**no reuse**

6.4.6 Imperfectly nested loops

Because of their simplicity, both *RefGroup* and *LoopCost* can also be applied to imperfectly nested loops. Consider the following example, where the first definition of A(J) is imperfectly nested:

```
DO J = 1, 100
    A(J) = 0
    DO I = 1, 100
        A(J) = A(J) + ...
```

RefGroup would place all references to A(J) in the same reference group. When we apply *RefCost* to calculate the number of cache lines accessed by a reference group, we need to select the most deeply nested member of the group. *LoopCost* then multiplies the result by the trip counts of all the loops that actually enclose the reference.

6.5 Loop permutation

The previous section presents our cost model for evaluating the data locality of a given loop structure with respect to cache. In this section we show how the cost model guides loop permutation to restructure a loop nest for better data locality.

A naive optimization algorithm would simply generate all legal loop permutations and select the permutation that yields the best estimated data locality using *LoopCost*. Unfortunately, generating all possible loop permutations takes time that is exponential in the number of loops and can be expensive in practice. It becomes increasingly unappealing when transformations such as strip mining introduce even larger search spaces.

Instead of testing all possible permutations, we show how our cost model allows us to design an algorithm to directly compute a preferred loop permutation.

6.5.1 Memory order

The locality evaluating function *LoopCost* does not calculate data reuse on outer loops; however, we can still restructure programs to exploit outer loop reuse. The key insight is that if loop l causes more reuse than loop l' when both are considered as innermost loops, l will also promote more reuse than l' when both loops are placed at the same outer loop position.

LoopCost can thus be considered to be a measure of the reuse carried by a loop. This allows us to select a desired permutation of loops called *memory order* that yields the best estimated data locality. We simply rank each loop l using *LoopCost*, ordering the loops from outermost to innermost $(l_1 \ldots l_n)$ such that $LoopCost(l_{i-1}) \geq LoopCost(l_i)$.

Memory order algorithm

The algorithm *MemoryOrder* is defined as follows. It computes *LoopCost* for each loop, sorts the loops in order of decreasing cache line accesses (*i.e.* increasing reuse), and returns this loop permutation.

Example

As an example, recall matrix multiply. We compute memory order with $cls = 4$. The reference groups for matrix multiply put the two references to C(I,J) in the same group on all the loops and A(I,K) and B(K,J) are placed in separate groups. *LoopCost* computes the relative reuse on each of the loops as seen below.

	LoopCost as innermost		
references	J	K	I
C(I, J)	$n * n^2$	$1 * n^2$	$1/4n * n^2$
A(I, K)	$1 * n^2$	$n * n^2$	$1/4n * n^2$
B(K, J)	$n * n^2$	$1/4n * n^2$	$1 * n^2$
totals	$2n^3 + n^2$	$5/4n^3 + n^2$	$1/2n^3 + n^2$

The algorithm *MemoryOrder* uses these costs to compute a preferred loop ordering of (J, K, I), from outermost to innermost. The same result is obtained by previous researchers [AK84, WL91].

6.5.2 Permuting to achieve memory order

We must now decide whether the desired memory order is legal. If it is not, we must select some legal loop permutation close to memory order. To determine whether a loop permutation is legal is straightforward. We permute the entries in the distance or direction vector for every true, anti, and output dependence to reflect the desired loop permutation. The loop permutation is illegal if and only if the first nonzero entry of some vector is negative, indicating that the execution order of a data dependence has been reversed [AK84, Ban90a, Ban90b, WL90].

In many cases, the loop permutation calculated by *MemoryOrder* is legal and we are finished. However, if the desired memory order is prevented by data dependences, we use a simple heuristic for calculating a legal loop permutation near memory order. The algorithm for determining this organization takes $max(D, n^2)$ time in the worst-case where n is the depth of the nest and D is the number of dependences, a definite improvement over considering all legal permutations, which is exponential in n. The algorithm is guaranteed to find a legal permutation with the desired inner loop, if one exists.

Permutation algorithm

Given a memory ordering $\{i_{\sigma_1}, i_{\sigma_2}, \ldots, i_{\sigma_n}\}$ of the loops $\{i_1, i_2, ..., i_n\}$ where i_{σ_1} has the least reuse and i_{σ_n} has the most, we can test if it is a legal permutation directly

---

Algorithm 6.3: **Determine the closest permutation to memory order**

NearbyPermutation $(\mathcal{O}, \mathcal{DV}, \mathcal{L})$

Input:

$\mathcal{O}$ $= \{i_1, i_2, ..., i_n\}$, the original loop ordering
$\mathcal{DV}$ = set of original legal direction vectors for l_n
$\mathcal{L}$ $= \{i_{\sigma_1}, i_{\sigma_2}, \ldots, i_{\sigma_n}\}$, a permutation of $\mathcal{O}$

Output:

$\mathcal{P}$ a nearby permutation of $\mathcal{O}$

Algorithm:

```
P = ∅ ;   k = 0 ;   m = n
while L ≠ ∅
  for j = 1, m
    l = l_j ∈ L
    if direction vectors for {p_1, ..., p_k, l} are legal
      P = {p_1, ..., p_k, l}
      L = L − {l} ;   k = k + 1 ;   m = m − 1
      break for
    endif
  endfor
endwhile
```

---

by performing the equivalent permutation on the elements of the direction vectors. If the result is a legal set of direction vectors, the loops are permuted accordingly.

Otherwise, we attempt to achieve a "nearby" permutation with the algorithm *NearbyPermutation*. The algorithm builds up a legal permutation in $\mathcal{P}$ by first testing to see if the loop i_{σ_1} is legal in the outermost position. If it is legal, it is added to $\mathcal{P}$ and removed from $\mathcal{L}$. If it is not legal, the next loop in $\mathcal{L}$ is tested. Once a loop l is positioned, the process is repeated starting from the beginning of $\mathcal{L} - \{l\}$ until $\mathcal{L}$ is empty. The following theorem holds for the *NearbyPermutation* algorithm.

Theorem 6.1 *If there exists a legal permutation where σ_n is the innermost loop, then NearbyPermutation will find a permutation where σ_n is innermost.*

The proof by contradiction of the theorem proceeds as follows. Given an original set of legal direction vectors, each step of the "for" is guaranteed to find a loop which results in a legal direction vector, otherwise the original was not legal [AK84, Ban90a].

In addition, if any loop σ_1 through σ_{n-1} may be legally positioned prior to σ_n it will be.

This characteristic is important because most data reuse occurs on the innermost loop and is due to spatial reuse, so positioning the inner loop correctly will yield the best data locality.

6.6 Data locality experimental results

We tested the algorithm for optimizing data locality independently and report some of these results here.

6.6.1 Matrix multiply

We executed all possible loop permutations of matrix multiply for 3 problem sizes, 150×150, 300×300 and 512×512, on a variety of uniprocessors to determine the accuracy of the *MemoryOrder* in predicting the best loop permutations. In Table 6.1, the permutations are ordered from the most desirable to the least based on the ranking computed by *MemoryOrder*. On all the processors, memory order JKI produced the best results in all but two cases. On all the processors but the Sequent, the entire ranking generally served to accurately predict relative performance. These results illustrate that *LoopCost* is effective in predicting relative reuse on outer loops as well as inner loops.

TABLE 6.1: **Matrix Multiply (in seconds)**

	Loop Permutation					
Processor	JKI	KJI	JIK	IJK	KIJ	IKJ
	150×150					
Sequent Weitek	26.0	27.1	31.1	30.7	28.4	26.9
Sun Sparc2	2.33	2.25	3.20	3.16	2.81	2.79
Intel i860	1.16	1.17	1.23	1.18	3.50	3.42
IBM RS6000	0.42	0.46	0.36	0.38	1.08	1.08
	300×300					
Sun Sparc2	18.3	17.8	26.1	25.2	24.9	27.1
Intel i860	9.7	10.2	21.7	21.8	59.1	58.9
IBM RS6000	3.37	3.47	12.5	12.5	56.4	56.5
	512×512					
Sun Sparc2	91.0	93.6	223	240	277	336
Intel i860	60.2	46.7	143	156	292	292
IBM RS6000	16.7	17.0	183	186	399	399

The disparity in execution times between permutations became greater as the processor speed increased. On the individual processors, execution times varied by significant factors of up to 3.69 on the Sparc2, 6.25 on the i860, and a dramatic 23.89 on the RS6000. These results indicate that data locality *should* be the overwhelming force driving scalar compilers today.

6.6.2 Stencil computations: Jacobi and SOR

Stencil computations such as Jacobi and SOR are finite difference techniques frequently used to solve partial difference equations [BHMS91]. Jacobi runs completely in parallel, while SOR causes a computational wavefront to sweep diagonally through the array. Both kernels were written using 500 $\times$ 500 2D arrays. We created and measured the execution time of the following program versions: (all of the actual programs appear in Figures 6.2 and 6.3).

Memory Order: Loops are ordered according to the algorithm *MemoryOrder*. Both temporal and spatial reuse are exploited. Neither program required the algorithm *NearbyPermutation*.

Poor Order: Loops are permuted in exactly the opposite manner as memory order. It is provided merely to show the worst-case performance if data locality is not taken into account.

FIGURE 6.2: **Stencil computation: Jacobi**

Memory Order
```
DO I = 2,N-1
    DO J = 2,N-1
        A(J,I) = 0.2*(B(J,I) + B(J-1,I) + B(J,I-1) + B(J+1,I) + B(J,I+1))
```
Poor Order
```
DO J = 2,N-1
    DO I = 2,N-1
        A(J,I) = 0.2*(B(J,I) + B(J-1,I) + B(J,I-1) + B(J+1,I) + B(J,I+1))
```
1D Tiles
```
DO JJ = 2,N-1,TILE
    DO I = 2,N-1
        DO J = JJ,MIN(JJ+TILE-1,N-1)
            A(J,I) = 0.2*(B(J,I) + B(J-1,I) + B(J,I-1) + B(J+1,I) + B(J,I+1))
```
2D Tiles
```
DO II = 2,N-1,TILE
    DO JJ = 2,N-1,TILE
        DO I = II,MIN(II+TILE-1,N-1)
            DO J = JJ,MIN(JJ+TILE-1,N-1)
                A(J,I) = 0.2*(B(J,I) + B(J-1,I) + B(J,I-1) + B(J+1,I) + B(J,I+1))
```

1D and 2D Tiles: The loops are first placed in memory order, then one or both loops are tiled in order to exploit reuse on the outer loop. This version shows that tiling can degrade performance if insufficient reuse exists on outer loops.

Memory order can be easily computed for both kernels. Temporal reuse is identical for both loops, since each results in the same number of reference groups (four for Jacobi, three for SOR). The J loop has much lower *LoopCost*, since all reference groups yield consecutive accesses when J is considered as the candidate innermost loop. In comparison, all references result in non-consecutive accesses with I innermost. Spatial reuse from consecutive accesses thus dominates when considering the proper loop permutation for good data locality.

The execution times measured for these program versions appear in Tables 6.2 and 6.3. Permuting loops to achieve memory order for these two kernels shows impressive improvements compared to their poorly ordered counterparts. The speedups for both programs range from a factor of 1.3 to 8.4. The RS6000 is especially sensitive to consecutive accesses; for Jacobi, its performance increased by a factor of 8.4 in memory order.

FIGURE 6.3: **Stencil computation: Successive Over Relaxation (SOR)**

Memory Order

```
    DO I = 2,N-1
        DO J = 2,N-1
            A(J,I) = 0.2*(A(J,I) + A(J-1,I) + A(J,I-1) + A(J+1,I) + A(J,I+1))
```

Poor Order

```
    DO J = 2,N-1
        DO I = 2,N-1
            A(J,I) = 0.2*(A(J,I) + A(J-1,I) + A(J,I-1) + A(J+1,I) + A(J,I+1))
```

1D Tiles

```
    DO JJ = 2,N-1,TILE
        DO I = 2,N-1
            DO J = JJ,MIN(JJ+TILE-1,N-1)
                A(J,I) = 0.2*(A(J,I) + A(J-1,I) + A(J,I-1) + A(J+1,I) + A(J,I+1))
```

2D Tiles

```
    DO II = 2,N-1,TILE
        DO JJ = 2,N-1,TILE
            DO I = II,MIN(II+TILE-1,N-1)
                DO J = JJ,MIN(JJ+TILE-1,N-1)
                    A(J-I,I) = 0.2*(A(J-I,I) + A(J-I-1,I) + A(J-I,I-1)
                                + A(J-I+1,I) + A(J-I,I+1))
```

Empirical results show that neither Jacobi nor SOR possess sufficient reuse at outer loops to justify tiling; spatial reuse is the most important factor for these stencil computations. In fact, tiling degrades performance. The tiled versions of each kernel begin to recover only as tile sizes become quite large.

Table 6.2: **Performance of Jacobi (in seconds)**

			1D Tiles			2D Tiles		
Processor	Memory	Poor	4	16	32	4x4	16x16	32x32
Sun Sparc2	0.37	0.75	0.48	0.40	0.39	0.47	0.40	0.38
Intel i860	0.12	0.48	0.30	0.16	0.14	0.23	0.14	0.13
IBM RS6000	0.09	0.76	0.26	0.13	0.11	0.12	0.09	0.09

Table 6.3: **Performance of SOR (in seconds)**

			1D Tiles			2D Tiles		
Processor	Memory	Poor	4	16	32	4x4	16x16	32x32
Sun Sparc2	0.31	0.41	0.41	0.33	0.32	0.39	0.33	0.32
Intel i860	0.20	0.57	0.37	0.24	0.21	0.27	0.21	0.21
IBM RS6000	0.13	0.41	0.22	0.16	0.14	0.15	0.13	0.13

6.6.3 Erlebacher

Erlebacher is a small benchmark program written by Thomas Eidson of ICASE that performs Alternating-Direction-Implicit (ADI) integration. It performs vectorized tridiagonal solves in each dimension, resulting in computation wavefronts across all three dimensions. Our results are for the forward and backward sweeps in Z dimension. The program versions we used are as follows.

Vector Order: All sequential loops (those carrying dependences) are placed outermost. Inner loops may all be executed in parallel. Within the parallel and sequential loop nests, each is ordered for data locality.

Parallel Order: All parallel loops are placed outermost. Inner loops are sequential and carry dependences corresponding to reuse. Within each group, loops are ordered for data locality. This is the loop permutation selected by optimizations that attempt to exploit temporal reuse without considering spatial reuse (consecutive accesses).

Memory Order: The loops are ordered according to descending *LoopCost*. As we have shown, both temporal and spatial reuse are considered when choosing memory order. All of the loop nests in Erlebacher were fully permutable, none required the algorithm *NearbyPermutation*.

Hand-coded: The loops are ordered according to the original source as provided by Thomas Eidson.

In addition, for each loop strategy we also have fused and separate versions of the program.

Alone: All the statements remain in the same nest as originally written, resulting in single statement loop nests.

Fuse: Each single statement loop nest is first permuted into the desired loop order. We then fuse all adjacent loop nests where legal. This version demonstrates the effects of exploiting reuse through loop fusion.

Figure 6.4 demonstrates vector, parallel, memory order and hand-coded versions for two of the loop nests found in the solution stage for the Z dimension. For this example, parallel order exploits the temporal reuse represented by dependences carried on the K loop. However as seen in Table 6.4, it results in the worst performance because it eliminates consecutive accesses for the references to array F.

Both vector and memory order place I as the innermost loop, resulting in cache line reuse for references to array F. However, memory order also selects the middle loop that yields the most reuse. In the first loop nest the J loop is preferred because both A(K) and B(K) become loop-invariant references, overcoming the savings derived for the K loop from putting F(I,J,K) and F(I,J,K-1) in the same reference group.

TABLE 6.4: **Performance of Erlebacher (in seconds)**

	Vector Order		Parallel Order		Memory Order		Hand
Processor	Alone	Fuse	Alone	Fuse	Alone	Fuse	Coded
Motorola 68020	836	838	847	848	838	840	841
Intel i386	20.5	20.6	20.2	20.1	19.9	19.8	20.1
Sequent Weitek	8.74	8.61	8.90	8.49	8.26	7.96	8.14
Sun Sparc2	1.09	1.07	.842	.682	.813	.672	.806
Intel i860	.705	.696	.660	.631	.548	.518	.547
IBM RS6000	.493	.480	.459	.441	.400	.383	.390

Figure 6.4: **Erlebacher: forward and backward sweeps in Z dimension**

```
{ Vector Order (outer loops sequential/inner loops parallel) }
DO K=N-2,1,-1
    DO J=1,N
        DO I=1,N
            F(I,J,K)=(F(I,J,K)-A(K)*F(I,J,K-1))*B(K)
DO K=N-2,1,-1
    DO J=1,N
        DO I=1,N
            F(I,J,K) = F(I,J,K) - C(K)*F(I,J,K+1) - E(K)*F(I,J,N)

{ Parallel Order (outer loops parallel/inner loops sequential) }
DO J=1,N
    DO I=1,N
        DO K=N-2,1,-1
            F(I,J,K)=(F(I,J,K)-A(K)*F(I,J,K-1))*B(K)
DO J=1,N
    DO I=1,N
        DO K=N-2,1,-1
            F(I,J,K) = F(I,J,K) - C(K)*F(I,J,K+1) - E(K)*F(I,J,N)

{ Memory Order (in order of decreasing LoopCost) }
DO K=N-2,1,-1
    DO J=1,N
        DO I=1,N
            F(I,J,K)=(F(I,J,K)-A(K)*F(I,J,K-1))*B(K)
DO J=1,N
    DO K=N-1,2,-1
        DO I=1,N
            F(I,J,K) = F(I,J,K) - C(K)*F(I,J,K+1) -E(K)*F(I,J,N)

{ Hand-coded }
DO K=N-2,1,-1
    DO J=1,N
        DO I=1,N
            F(I,J,K)=(F(I,J,K)-A(K)*F(I,J,K-1))*B(K)
DO J=1,N
    DO K=N-2,1,-1
        DO I=1,N
            F(I,J,K) = F(I,J,K) - C(K)*F(I,J,K+1) - E(K)*F(I,J,N)
```

In comparison, in the second loop nest K is the preferred middle loop. By combining F(I,J,K) and F(I,J,K+1) in the same reference group and making F(I,J,N) a loop-invariant reference, the K loop provides more savings than the J loop can by making both A(K) and B(K) loop-invariant. This results in approximately a 5-10% improvement for the i860, showing that simply selecting the correct innermost loop is not enough sufficient to yield the maximum data locality.

Our results show that data locality grows in importance with processor speeds. Loop fusion provides additional improvements in execution time, but selecting the correct loop permutation for good data locality again yields the greatest benefit.

6.7 Parallelism

In the following two subsections, parallelism is evaluated and exploited. We first present a performance estimator that evaluates the potential benefit of parallelism. A parallel code generation strategy then uses performance estimation and the cost model developed in the previous section with other transformations to combine effective parallelism and memory order, making tradeoffs as necessary.

6.7.1 Performance estimation

This section uses performance estimation to quantify the effects of parallelism on execution time. Our performance estimator predicts the cost of parallel and sequential performance using a loop model and a *training set* approach.

The goal of our performance estimator is to assist in code generation for both shared and distributed memory multiprocessors [BFKK92, KMM91]. Modeling the target machines at an architectural level would require calculating an analytical model for each supported architecture. Instead our performance estimator uses a training set to characterize each architecture in a machine-independent fashion. A training set is a group of kernel computations that are compiled, executed and timed on each target machine. They measure the cost of operations such as multiplication, branching, intrinsics, and loop overhead. These costs are then made available to the performance estimator via a table of data. Note, the training sets for the performance estimator only measure access times to data in registers or the closest cache.

Of particular interest is the estimation of parallel loops. Given sufficient parallel granularity, using all available processors results in the best execution time.

Estimating the cost in this circumstance may be modeled by determining the following.

$$\begin{aligned} c_s = & \text{ cost of starting parallel execution} \\ c_f = & \text{ cost of forking and synchronizing} \\ & \text{ a parallel process} \\ P = & \text{ number of processors} \\ b = & \text{ number of iterations of the parallel loop} \\ t(B) = & \text{ cost of the loop body} \end{aligned}$$

If the loop bounds are unknown, a guess is used that is based on the declared dimension of the arrays accessed in the loop. With these parameters the performance of a parallel loop with sufficient work may be estimated by:

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

However, if the amount of work is not sufficient, parallel loop execution is more difficult to model. Instead of an equation, a table is used to indicate the appropriate number of processors for the best performance. The model and the table are generated using a training set.

The sample training set for determining parallel loop overhead begins by varying the total amount of work. For each unit of work, the number of processors is varied from 1 to the total available. The number of processors which minimize the execution time of this work is selected. The result of a training run for parallel loops on the Sequent S81 appears in Figure 6.5.

This particular training run repeatedly performed a single scalar operation that executed for approximately 10 microseconds, which represents one unit of work in Figure 6.5. Each of the contour lines indicates a particular execution time. The single line cutting across the contour lines represents the minimum execution time for executing a particular work load and the appropriate number of processors. When total work is below 250 a table determines the appropriate number of processors and approximate execution time. Once the total work is over about 250, the parallel loop model is used. The estimator provides a single cost function for evaluating loops that chooses between the techniques based on total work and number of loop iterations.

$Estimate(l, how)$ returns $\langle \tau, np \rangle$ where
l is a loop with body B
how indicates whether l may be run in parallel

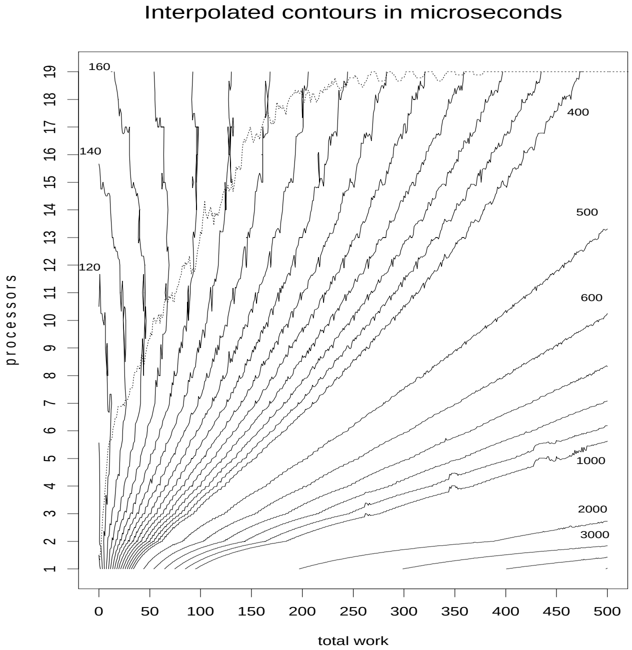


FIGURE 6.5: **Parallel loop training set**

This function returns a tuple $\langle \tau, np \rangle$ with an estimate τ which is the minimal execution time and the number of processors np necessary to obtain the estimate, based on whether the loop is parallel. Note, if the loop is sequential or it is not profitable to run it in parallel, the sequential running time and $np = 1$ are returned.

6.7.2 Introducing parallelism

The key to introducing parallelism is to maintain memory order during parallelization by using strip mining and *loop shifting* (loop shifting moves an inner loop outward across one or more loops). Strip mining performs two functions in parallelization. (1) It preserves cache line reuse in parallel execution. Without strip mining, consecutive iterations may be scheduled on different processors, denying cache line reuse. (2) Because strip mining results in two loops, the parallel iterator loop may be shifted outward to maximize granularity while the sequential strip remains in place providing the data locality introduced using memory order. To illustrate this point, consider the subroutine *dmxpy* from Linpackd written in memory order [DBMS79].

```
DO J = JMIN, N2
    DO I = 1, N1
        Y(I) = Y(I) + X(J) * M(I,J)
```

The J loop is not parallel. The I loop can be parallel. Both contain reuse. A simple parallelization that maximizes granularity would interchange the two loops and make the I loop parallel without strip mining. Unfortunately with this organization, the parallel loop may be scheduled such that consecutive iterations are assigned to different processors causing false sharing of Y and eliminating cache line reuse for consecutive accesses to X and M. In addition, cache lines containing the same array elements would be required at multiple processors, increasing total memory and bus utilization.

We instead strip mine a parallel loop by strip size SS $=\lceil$N1/P$\rceil$ to provide reuse on the strip and parallelize the resultant iterator. If the parallel loop is outermost, as in matrix multiply, parallelization is complete. If not, we use loop shifting to move the parallel iterator to its outermost legal position, maximizing its granularity. Applying this strategy to *dmxpy*, we begin with the memory ordered loop nest. The I loop is the only parallel loop and it contains reuse. Therefore, it is strip mined. The parallel iterator is not outermost, but it is legally shifted to the outermost position. The compiler shifts the loop, resulting in maximum granularity and data locality as illustrated below.

```
PARALLEL DO I = 1, N1, SS
      DO J = JMIN, N2
            DO II = I, MIN(I + SS - 1, N1)
                  Y(II) = Y(II) + X(J) * M(II,J)
```

6.7.3 Strip mining

If a loop is selected to be performed in parallel, it is strip mined if it contains any reuse. Given sufficient iterations, strip mining exploits data locality and parallelism by using $\lceil$N/P$\rceil$ as the strip size where N is the number of iterations. Assuming $cls \leq$ P, the iteration space is sufficiently large if P $<$ N. If

$$\mathrm{P} < \mathrm{N} < cls * \mathrm{P},$$

strip mining by $\lceil$N/P$\rceil$ is less than the *cls* and may result in false sharing. However, the granularity of the parallel loop does match P and some reuse will occur. In this case, we still strip mine by $\lceil$N/P$\rceil$. However, if N $<$ P, strip mining may provide reuse but at the cost of drastically reducing the granularity of parallelism. This tradeoff is very machine specific. We choose not to strip mine when N $<$ P.

When memory order is computed, the loops are marked to indicate if they contain any reuse. If there is reuse, the strip mining algorithm uses the above equations to select a strip size that maximizes granularity and reuse. If there is no reuse, the strip mining algorithm does not perform strip mining, giving more flexibility to the scheduler.

6.7.4 Parallelization algorithm

For memory ordered loop nests that are not parallel on the outermost loop, the *Parallelization* algorithm uses loop shifting to introduce parallelism. It uses loop shifting, rather than a general loop permutation algorithm, in order to minimize the effect of parallelization on data locality. It performs strip mining when the loop contains reuse before shifting for the same reason. In the worst case, it is O(n^2) time.

Algorithm 6.4 introduces parallelism into memory order. It begins by testing whether the outermost loop is parallel. In the first iteration of the "for k" ($j = k = 1$),

---

ALGORITHM 6.4: **Introduce parallelism**

Parallelization ($\mathcal{L}$)
  INPUT: $\mathcal{L} = \{\sigma_1, \ldots, \sigma_n\}$ a legal permutation
  OUTPUT: $\mathcal{T}$ a parallelizing transformation
  ALGORITHM:
    $\mathcal{T} = \emptyset$
    for $j = 1, n$
      for $k = j, n$
        if σ_k legal at position j & parallel
          $\mathcal{T} = \{$ **StripMine**(σ_k),
              shift iterator to j, parallelize it $\}$
          return $\mathcal{T}$
        elseif σ_k legal at j & σ_j becomes parallel
          $\mathcal{T} = \{$**StripMine**(σ_k), *shift k iterator to j,*
              StripMine(*new* σ_{j+1}),
              parallelize the j+1 iterator $\}$
        endif
      endfor
      if $\mathcal{T} \neq \emptyset$ **return** $\mathcal{T}$
    endfor

---

the first "if" tests if the outermost loop is parallel. Trivially, a shift of loop σ_j to position j is always legal.

If the loop is parallel, it is strip mined and parallelized and the algorithm returns. If the loop is not parallel, a legal shift of an inner loop to position j which is parallel at position j is sought. If a parallel loop is found that can be shifted outermost to j, it is strip mined, parallelized and shifted and the algorithm returns. Otherwise, a shift to position j may cause the next inner loop, *i.e.* the loop originally positioned at j, to be parallel. This situation is determined in the "elseif." Because it is more desirable to parallelize a loop at position j than at $j+1$, all other shifts to position j are considered before this parallelization is returned at the completion of the "for k."

In Algorithm 6.4 *Parallelize* does not detect when strip mining results in a strip size of less than *cls* or strip mining is not performed due to insufficient parallel iterations. As we saw in Section 6.3 these conditions are unavoidable in some cases and the best possible performance is gained even when they hold. However, we extend *Parallelize* as follows to seek a better parallelization for which neither condition holds.

If *StripMine* returns with a strip size of less than *cls* or does not strip mine due to insufficient parallel iterations, then the number of parallel iterations PI and the size of the strip SS are recorded and the "for k" loop continues instead of returning. If the "for k" finds a parallelization where neither condition holds, it returns. Otherwise, at the completion of the "for k" it selects the parallelization with the largest pair (PI, SS).

6.8 Optimization algorithm

The optimization driver for exploiting data reuse and introducing parallelism appears in Algorithm 6.5. It combines the component algorithms described in the previous sections and is also $O(n^2)$ time.

It first calls *MemoryOrder* to optimize data locality via loop permutation. It then determines whether the loop contains sufficient computation to pursue parallelism. If it does, the memory ordered loop nest is provided to the algorithm *Parallelize*. If needed, *Parallelize* uses strip mining and loop shifting to introduce loop level parallelism.

The search space in *Parallelize* is constrained to meet our goal of perturbing the memory order as little as possible. If parallelism is not discovered and would be

ALGORITHM 6.5: **Optimizing for parallelism and data locality**

SimpleOptimizer ($\mathcal{L}$)
INPUT: $\mathcal{L} = \{l_1, \ldots, l_n\}$
OUTPUT: $\mathcal{T}$ an optimization of $\mathcal{L}$
ALGORITHM:
$\mathcal{O}$ = **MemoryOrder**($\mathcal{L}$)
np = **Estimate** ($\mathcal{O}$, *parallel*)
if $np > 1$ (parallelism is profitable)
$\mathcal{T}$ = **Parallelize**($\mathcal{O}$)
endif
perform { $\mathcal{O}, \mathcal{T}$ }

profitable, other optimization strategies that consider all loop permutations, loop skewing [WL90], or loop distribution (see Chapter 7) should be explored.

6.9 Experimental results

The overall parallelization strategy was also tested by applying it by hand to two kernels. The results of these experiments and those for data locality are very promising.

6.9.1 Matrix multiply

The speed-ups of a parallel tiled matrix multiply on 7 and 19 processors of a Sequent Symmetry S81 for arrays of size 150×150 and 300×300 are presented in Table 6.5. We ran a sequential version with the loops in memory order JKI, a sequential tiled version, and the identically tiled parallel version. The parallel version is tiled by 4 and is the same version presented in Section 6.1. Besides tiling, no other low-level memory optimizations were used. The speed-ups were basically linear for both matrix sizes when comparing the two tiled versions.

TABLE 6.5: **Speed-ups for Parallel Matrix Multiply**

	speed-up of parallel JKI tiled			
	19 processors		7 processors	
	over sequential JKI	over sequential JKI tiled	over sequential JKI	over sequential JKI tiled
150x150	20.5	18.8	7.5	6.8
300x300	20.1	18.7	7.5	7.0

6.9.2 Dmxpy

The subroutine *dmxpy* from Linpack was optimized using these algorithms as illustrated in Section 6.7.2. In scientific programs, there are many instances of this type of doubly-nested loop which iterates over vectors and/or matrices, where only one loop is parallel and it is best ordered at the innermost position. These loops may be an artifact of a vectorizable programming style. They appear frequently in the Perfect benchmarks [CKPK90], the Level 2 BLAS [DCHH88], and the Livermore loops [McM86].

Table 6.6 illustrates the performance benefits with the organization of *dmxpy* generated by our algorithm on matrices of size 200×200 on 19 processors. For comparison, the performance when the I strip is not returned to its best memory position and a parallel inner I loop were also measured.

TABLE 6.6: **Dmxpy on 19 processors**

	loop organization I loop parallel		
	I J II	I II J	J I
speed-up over sequential JI	16.4	13.8	2.9

6.10 Related work

Our work bears the most similarity to research by Wolf and Lam [WL91]. They develop an algorithm that estimates all temporal and spatial reuse for a given loop permutation, including reuse on outer loops. This reuse is represented as a *localized vector space*. Vector spaces representing reuse for individual and multiple references are combined to discover all loops $\mathcal{L}$ carrying some reuse. They then exhaustively evaluate all legal loop permutations where some subset of $\mathcal{L}$ is in the innermost position, and select the one with the best estimated locality.

Wolf and Lam's algorithm for selecting a loop permutation is potentially more precise and powerful than the one presented here. It directly calculates reuse across outer loops and can suggest loop skewing and reversal to achieve reuse; however, how often these transformations are needed is yet to be determined. Skewing in particular is undesirable because it reduces spatial reuse.

Gannon *et al.* also formulate the dependence testing problem to give reuse and volumetric information about array references [GJG88]. This information is then used to tile and interchange the loop nests for cache, after which parallelism is inserted at the outermost possible position. They do not consider how the parallelism affects the volumetric information nor if interchange would improve the granularity of parallelism.

Porterfield presents a formula that approximates the number of cache lines accessed for a loop nest, but it is restricted to a cache line size of one and loops with uniform dependences [Por89]. Ferrante *et al.* present a more general formula that also approximates the number of cache lines accessed and is applicable across a wider range of loops [FST91]. However, they first compute an estimate for every array reference and then combine them, trying not to do dependence testing. Like Wolf and Lam, they exhaustively search for a loop permutation with the lowest estimated cost.

Many algorithms have been proposed in the literature for introducing parallelism into programs. Callahan *et al.* use the metric of minimizing barrier synchronization points via loop distribution, fusion and interchange for introducing parallelism [ACK87, Cal87]. Wolf and Lam [WL90] introduce all possible parallelism via the unimodular transformations: loop interchange, skewing, and reversal. Neither of these techniques try to map the parallelism to a machine, or try take into account data locality, nor is any loop bound information considered. Banerjee also considers introducing parallelism via unimodular transformations, but only for doubly nested loops [Ban90b]. Banerjee does however consider loop bound information.

Because we accept some imprecision, our algorithms are simpler and may be applied to computations that have not been fully characterized in Wolf and Lam's unimodular framework. For instance, we can support imperfectly nested loops, multiple loop nests, and imprecise data dependences. We believe that this approximation is a very reasonable one, especially in view of the fact that we intend to use a scalar cache tiling method as a final step in the code generation process [CCK90]. In addition, the algorithms presented here are $O(n^2)$ time in the worst case, where n is the depth of the loop nest, and are a considerable improvement over work which compares all legal permutations and then picks the best, taking exponential time. Our approach has appeared elsewhere [KM92].

6.11 Discussion

We have addressed the problem of choosing the best loop ordering in a nest of loops for exploiting data locality and for generating parallel code for shared-memory multiprocessors. As our experimental results bear out, the key issue in loop order selection is achieving effective use of the memory hierarchy, especially cache lines. Our approach improves data locality, provides the highest granularity of parallelism, and properly positions loops for low-level memory optimizing transformations. When possible, the benefits of parallelism and data locality are therefore both exploited.

The next chapter incorporates this algorithm into a more general, interprocedural approach for generating parallel code.

Chapter 7

An Automatic Parallel Code Generator

In this chapter, we present a parallel code generation algorithm for shared-memory multiprocessors. We use the results of Chapters 4 and 5 to design an interprocedural algorithm for complete application programs. The key parallelization component is the algorithm for improving data locality and introducing parallelism developed Chapter 6. This chapter presents a new technique that performs loop fusion to enhance granularity. When necessary, partial parallelism is exploited using loop distribution. A general, unified treatment of fusion and distribution for loop nests is described and shown optimal under certain constraints. The result is a cohesive loop-based, interprocedural parallelization algorithm which builds on extends the work in previous chapters.

7.1 Introduction

The goal of the optimization algorithm presented in this chapter is to introduce parallelism in a way that minimizes execution time over the entire program. The major components used by the parallelizer to discover and exploit parallelism and data locality are as follows.

1. Loop-based intraprocedural transformations
   - loop permutation
   - strip mining
   - loop fusion
   - loop distribution
2. Interprocedural transformations
   - loop embedding
   - loop extraction
   - procedure inlining
   - procedure cloning
3. Performance estimation

The purpose of this work is to improve execution time by exploiting and discovering parallelism and improving data locality. Exploiting parallelism takes three forms: (1) assuring sufficient granularity to make parallelism profitable, (2) maximizing granularity, making each parallel task as large as possible, and (3) matching the number of parallel iterations to the machine. Assuring sufficient granularity and matching it to the machine is dependent on the architecture. Most previous research focuses on discovering parallelism and/or maximizing its granularity without regard to data locality [Cal87, WL90, ABC+87].

Our approach addresses all of these concerns. The key component is the combination of loop interchange and strip mining developed in Chapter 6 which considers all these factors. Sufficient granularity is assured using performance estimation (see Section 6.7.1). In this chapter, we present a loop fusion algorithm to further increase granularity. In addition, we use loop distribution to discover parallelism when necessary. A unified treatment of fusion and distribution shows the problems to be identical. This algorithm is shown to be optimal for a restricted problem scope.

This chapter uses the extensions developed in Chapter 5 to design an algorithm that considers loop-based transformations even in the presence of procedure calls. The loop-based transformations determine which, if any, interprocedural transformations are necessary. The interprocedural transformations are called *enablers* and are applied using goal-directed interprocedural optimization [BCHT90]. The interprocedural transformations are applied only when they are expected to improve execution time; *i.e.* the interprocedural transformations are required to perform a loop-based parallelization of a loop nest that spans procedures.

7.2 Parallel code generation

7.2.1 Driving code generation

The driver for optimizing a program appears in Algorithm 7.1. The algorithm *Driver* optimizes routines and loops in reverse postorder using the augmented call graph G_{ac}, guaranteeing that a procedure or loop is optimized only when the context of its calling procedures and outer loops is known. This formulation is general enough for performing many whole program optimizations on programs without recursion, for which G_{ac} is a DAG. It is used here to introduce a single level of high-granularity parallelism and exploit the memory hierarchy. We illustrate this algorithm by optimizing the following example.

EXAMPLE 7.1:

```
PROCEDURE C
      CALL S
      DO I = 1,N
            CALL S
      ENDDO

PROCEDURE S
      DO J = 1, N
            ...
      ENDDO
```

edges $\in$ G_{ac}:
(C, S) – call edge
(C, I) – loop edge
(I, S) – call edge
(S, J) – loop edge

Driver begins with C and marks it visited. It then tests if all of procedure C's predecessors have been optimized. Let's assume none of them have. *Driver* then tries to recursively optimize each of C's successors in the "forall." The first successor is the call to S. However, all of S's predecessors have not been visited so it proceeds to the next successor, the I loop, whose predecessors have all been visited. *Driver* then optimizes the I loop in step (3) of the algorithm using *Optimize* to specify a loop-based parallelization. Let's assume *Optimize* simply parallelizes the I loop. The procedure *Transform* applies this parallelization and marks all the descendants of the I loop (S and J) as optimized. Note, the descendants are *not* marked visited.

Driver continues to seek optimization candidates for the descendants of the I loop to ensure that all paths to a procedure are optimized if possible. *Driver* is called again on procedure S, this time all of S's predecessors have been visited. It checks to see if all the predecessors of S have been optimized as well as visited. Of S's predecessors, C and the I loop, only the I loop has been optimized. Therefore, there is a calling sequence to S that does not contain parallelism and S should be and is considered for optimization. The only successor of S is the J loop and it is optimized next using *Optimize*. If a parallelization is specified, it is performed. There are no more edges or unvisited nodes, consequently the algorithm terminates.

Discussion

This section details more formally the *Driver* algorithm. *Driver* is initially called on the program's root node, and each node in the G_{ac} is initialized to be *unvisited* and *unoptimized*. The G_{ac} is traversed in reverse postorder, such that a node n is only visited once all its predecessors are visited. Note, that only procedure nodes may have

Algorithm 7.1: **Driver for parallel code generation**

Driver (n)
Input: n is a node in the G_{ac}
Algorithm:
if any predecessor of n is not visited **return**
mark n *visited*
if n is a procedure and all predecessors of n are marked *optimized* **return**
forall m where $\exists$ edge (n, m) in topological order
(1) **if** m is a procedure
Driver (m)
(2) **if** (m is *visited*) and ($\exists$ $(m, s) \in \mathrm{G}_{ac}$ *such that* s is *optimized*)
$\mathcal{T}$ = **Fusion**$(s_1, \ldots, s_k)$
Transform$(n, \{s_1, \ldots, s_k\}, \mathcal{T})$
endif
(3) **else**
if m is an outer loop of procedure n
let $\mathcal{L}$ include the entire looping structure rooted at m
$\mathcal{T}$ = **Optimize**$(\mathcal{L})$
Transform$(n, m, \mathcal{T})$
endif
Driver (m)
endif
endforall

more than one immediate predecessor. If n is a procedure and all its predecessors are marked as *optimized*, then parallelism has been introduced in each calling context and the procedure does not need to be optimized for additional parallelism.

Loop nests are optimized as a whole when the outermost loop of the nest is encountered in step (3) of the algorithm. If m is the outermost loop of a nesting structure $\mathcal{L}$ at step (3), $\mathcal{L}$ is constructed and optimization is performed on it. If optimization is successful, the transformation is applied and all the affected nodes are marked as *optimized* by the subroutine *Transform* which appears in Algorithm 7.2. However, the algorithm continues to recurse over all successors of a node regardless of node type until all nodes have been visited, or all the callers of a procedure have been optimized, or the loop or procedure itself has been optimized.

At step (2), if m is a procedure and m is visited, optimization has just been performed on all the loops and calls it contains. If any of these successors of m are optimized, they are now considered for fusion. This feature allows fusion of loops that are not enclosed by an outer loop. Fusion of loops that are enclosed by an outer loop and all other loop-based transformations are determined by the *Optimize* algorithm.

For example, consider a procedure P containing two adjacent loops, l_1 and l_2. The procedure P is visited followed by l_1 and l_2. Both are determined to be parallel and are marked as *optimized*. Then *Fusion* (l_1, l_2) is considered to maximize granularity and to reduce synchronization and loop overhead.

In order to simplify the discussion, this version of *Driver* does not perform any interprocedural transformations. The parallelizer is extended in Section 7.6 to perform these interprocedural transformations. This version of the parallelizer does and must perform procedure cloning for correctness.

7.2.2 Procedure cloning

Procedure cloning is required when optimizations along distinct call paths want to use different versions of the same procedure [CKT86a, CHK92]. In our algorithm, this situation only occurs when a procedure is called more than once.

Consider again Example 7.1. Procedure S may be called once directly and once from inside the I loop. The second call to S is parallelized by making the I loop parallel and the first call by making the J loop parallel. In this case, two versions of S are needed. The original sequential version of S is required for the parallel I loop. Another version, called the clone, is needed in which to specify the J loop be performed in parallel.

To determine if cloning is necessary, the parallelizer keeps track of whether or not parallelism has been introduced in callers using the *Walk&MarkG*$_{ac}$ algorithm. The subroutine *Transform* in Algorithm 7.2 determines if a clone is necessary.

Remember, only nodes whose predecessors have all been visited, where at least one is unoptimized, are candidates for optimization in *Parallelization*. If *Optimize* returns a parallelizing transformation for a candidate, *Transform* is called to apply it to the appropriate procedure n. If none of n's predecessors are optimized, a clone is unnecessary and the transformation is performed directly on the procedure n. If any of n's predecessors are marked *optimized*, then a clone is need on which to apply the current optimizing transformation. If a clone already exists, then other loop nests in the procedure have been optimized and the new optimization is also applied to this existing clone. Otherwise a clone is created, and the optimization is applied to

ALGORITHM 7.2: **Applying transformations and cloning**

Transform $(n, \mathcal{L}, \mathcal{T})$
INPUT: n is a procedure node in the G_{ac}
$\mathcal{L}$ is a set of loops in n
$\mathcal{T}$ a transformation to perform on $\mathcal{L}$
ALGORITHM:
if $\mathcal{T} = \emptyset$ **return**
if $\exists$ a predecessor of n marked *optimized*
if $\nexists$ n_{clone} **create** a clone of procedure n, n_{clone}
Apply $(n_{clone}, \mathcal{T})$
else
Apply $(n, \mathcal{T})$
endif
forall $l \in \mathcal{L}$ **Walk&MarkG**$_{ac}(l)$

Walk&MarkG$_{ac}$ (n)
INPUT: n is a node in the G_{ac}
ALGORITHM:
if n marked optimized **return**
mark n optimized
forall m where $\exists$ edge (n, m) in topological order
Walk&MarkG$_{ac}(m)$
endforall

the cloned procedure. Only two versions of any procedure are required using this simplified algorithm.

7.2.3 Loop-based optimization

The procedure *Optimize* in Algorithm 7.3 determines an optimizing transformation to parallelize an arbitrary loop nest $\mathcal{L} = \{l_1, \ldots, l_n\}$. $\mathcal{L}$ describes a loop nesting structure rooted at l_1 that may contain constructs such as imperfectly nested loops, procedure calls, and control flow. The loop-level transformations *Optimize* considers are loop permutation, strip mining, loop fusion, and loop distribution.

Optimize first calls *Order&Parallelize* which performs loop permutation and strip mining using the algorithms developed in Chapter 6 for improving data locality and exploiting parallelism. *Order&Parallelize* begins by improving data locality using *MemoryOrder*. Given sufficient granularity, it then inserts parallelism into the resultant nest using the algorithm *Parallelize*. *MemoryOrder* and *Parallelize* are defined

---

ALGORITHM 7.3: **Optimizing a loop nest**

Optimize ($\mathcal{L}$)
  INPUT: $\mathcal{L} = \{l_1, \ldots, l_n\}$, a loop nest
  OUTPUT: $\mathcal{T}$ a parallelizing transformation
  ALGORITHM:
(1) $\mathcal{T}$ = **Order&Parallelize** ($\mathcal{L}$)
  if ($\mathcal{T} \neq \emptyset$) **return** $\mathcal{T}$
(2) $\mathcal{BD}$ = **BreakDependences** ($\mathcal{L}$)
  if $\mathcal{BD} \neq \emptyset$
    $\mathcal{T}$ = **Order&Parallelize** ($\mathcal{BD}(\mathcal{L})$)
    if ($\mathcal{T} \neq \emptyset$) **return** $\{\mathcal{T}, \mathcal{BD}\}$
  endif
(3) $\mathcal{T}$ = **Distribution**($\mathcal{BD}(\mathcal{L})$)
  return $\{\mathcal{T}, \mathcal{BD}\}$

Order&Parallelize ($\mathcal{L}$)
  INPUT: $\mathcal{L} = \{l_1, \ldots, l_n\}$, a loop nest
  OUTPUT: $\mathcal{T}$ a parallelizing transformation
  ALGORITHM:
  $\mathcal{O}$ = **MemoryOrder**($\mathcal{L}$)
  np = **Estimate** ($\mathcal{O}(\mathcal{L})$, *parallel*)
  if $np = 1$ (parallelism is not profitable) **return** $\mathcal{O}$
  $\mathcal{T}$ = **Parallelize**($\mathcal{O}(\mathcal{L})$)
  if $\mathcal{T} \neq \emptyset$ (parallelism found)
    $\mathcal{F}$ = **Fusion** ($\mathcal{T}(\mathcal{O}(\mathcal{L}))$)
    return $\{ \mathcal{O}, \mathcal{T}, \mathcal{F} \}$
  endif
  return $\emptyset$

---

in Chapter 6. If *Order&Parallelize* has been successful at introducing parallelism at some level, fusion of loops in the resultant nest is considered. This algorithm is discussed in detail below.

If step (1) is unsuccessful, step (2) attempts to satisfy as many dependences as possible with *BreakDependences*. The literature includes a collection of transformations that are used to satisfy specific dependences that inhibit parallelism. They include loop peeling, scalar expansion [KKLW80a], array renaming [AK87, KKLW80a], alignment and replication [Cal87], and loop splitting [AK87].

These transformations may introduce new storage to eliminate storage-related anti or output dependences, or convert loop-carried dependences to loop-independent dependences, often enabling the safe application of other transformations. If all the dependences carried on a loop are eliminated, the loop may then be run in parallel. If *BreakDependences* has some success, *Order&Parallelize* is called again on the result. If neither step (2) or (3) is successful, loop distribution to introduce partial parallelism is considered, as discussed in the next section.

The *Optimize* algorithm is able to introduce parallelism into loops which contain procedure calls and unstructured control flow. This important ability stems from the analysis and transformations described in Chapters 4 and 5.

7.3 Partitioning for loop distribution and loop fusion

In this section, we present a new algorithm which maximizes parallelism and minimizes the number of loops. It unifies the treatment of loop distribution and fusion. This solution is shown optimal for a single loop under certain constraints.

Loop distribution is safe if the partition of statements into new loops preserves all of the original dependences (see Section 4.2). Dependences are preserved if any statements involved in a dependence recurrence are placed in the same loop. The dependences between the partitions then form an directed acyclic graph that can always be ordered using topological sort [AK87, KKP+81].

By first choosing a safe partition of the loops with the finest possible granularity and then fusing partitions back together larger partitions may be formed. This transforms the loop distribution to one of loop fusion, a problem thought to be very hard. In fact, Goldberg and Paige prove a nearby fusion problem NP-Hard, but their result is not applicable here [GP84]. The algorithm we present is linear.

In the loop fusion problem for parallelism, the partitions begin as separate loops that are either parallel or sequential and may or may not be legally fused together. In loop distribution, it is always legal to fuse the loops back together and run the original loop sequentially. We would like to utilize all the parallelism and have the fewest loops. Callahan refers to these criteria as *maximal parallelism* with *minimum barrier synchronization* [Cal87].

The loop distribution and loop fusion problem is a graph partitioning problem on a directed acyclic graph (DAG). Each node in the graph represents a sequential or parallel loop containing a set of statements. There are data dependence edges, some of which are fusion preventing. Fusion preventing edges exist between nodes that cannot be fused together without changing the loop's semantics. Fusion preventing edges also exist between parallel nodes that, when fused, force the resultant loop to be sequential. We seek to minimize the number of partitions (loops), subject to the constraint that a particular partition contains only one type of node; *i.e.*, undirected fusion preventing edges are implicit between sequential and parallel nodes. A more formal description follows.

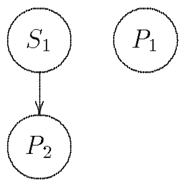
Partitioning problem:

> **Goal:** group parallel loops together and sequential loops together such that parallelism is maximized while minimizing the number of partitions.
>
> **Given a DAG with:**
>
> | | |
> |---|---|
> | nodes | parallel & sequential loops |
> | edges | data dependence edges some of which are fusion preventing edges |
>
> **Rules:**
> 1. cannot fuse two nodes with a fusion preventing edge between them
> 2. cannot change relative ordering of two nodes connected by an edge
> 3. cannot fuse sequential and parallel nodes

Callahan presents a greedy algorithm for a closely related problem that omits rule 3 [Cal87, ACK87]. His work also tries to partition a graph into minimal sets, but his model of parallelism includes loop-level and fork-join task parallelism. For example, consider the example graph in Figure 7.1.

Callahan's greedy algorithm partitions this graph into $\{P_1,S_1\}$ and $\{P_2\}$, and places a barrier synchronization between the partitions. S_1 and P_1 run in parallel with each other, and the iterations of P_1 may be performed in parallel. P_2 is performed in parallel once they both complete. Callahan's formulation of the loop distribution problem ignores the node type, enabling the greedy algorithm to provably minimize the number of loops and maximize parallelism for a single level-level [Cal87, ACK87]. Our model of parallelism differs in that it only considers loop-level parallelism. If

FIGURE 7.1: **Counter example for the greedy algorithm**



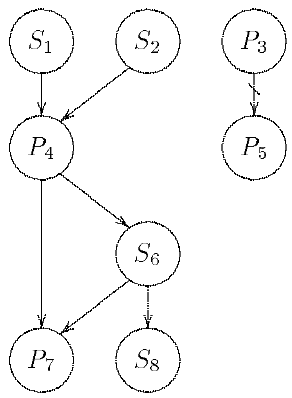
parallel the cost of loop overhead is higher than the cost of barrier synchronization, our model will be an improvement.

Consider using the greedy algorithm for the loop distribution and loop fusion problem restricted to loop-level parallelism. In the example in Figure 7.1, the loops S_1 and P_1 are of different node types and cannot be placed in the same partition, therefore one of P_1 or S_1 must be selected. If S_1 is selected first, the greedy partition is $\{S_1\}$, $\{P_1, P_2\}$. If P_1 is selected first, the greedy partition is $\{P_1\}$, $\{S_1\}$, $\{P_2\}$. The greedy algorithm is foiled because it cannot determine which node to select first. Note, if the graph consists of just sequential nodes or just parallel nodes, then this is equivalent to Callahan's problem formulation. Therefore, the greedy algorithm is optimal in the number of loop nests created when the nodes are only of one type. Our algorithm is based on this observation.

7.3.1 Simple partition algorithm

Our solution divides the problem into two parts, a sequential graph and a parallel graph. The greedy algorithm then minimizes the number of partitions for each of the graphs and maximizes parallelism in the parallel graph. Because parallel nodes and sequential nodes cannot be placed in the same partition without violating the maximum parallelization constraint, merging the solutions will result in a partitioning that maximizes parallelism and minimizes the number of loops. The remainder of this section describes how to correctly and efficiently construct the separate graphs and merge the reduced, partitioned solutions back together.

FIGURE 7.2: **Partition graph G_o**



Correctness of problem division

To separate the problem into two parts, the essential relationships in the original graph G_o must be preserved in the two component graphs, the sequential graph G_s and the parallel graph G_p, without introducing unnecessary constraints. First, all the ordering edges between two nodes of the same type must be preserved in the component graphs. In addition, G_o may represent relationships that prevent nodes of the same type from being in the same partition, but that do not have an edge between them. Consider G_o in Figure 7.2. Although an edge does not directly connect S_1 and S_6, they may not be fused together without including P_4. This fusion would violate the maximal parallelism constraint. These transitive, fusion preventing relationships are required between two nodes of the same type, that are connected by a path of nodes of a different type. No other edges are required. G_s also contains edges from G_o that connect sequential nodes (likewise for G_p).

The simplest way to preserve all the fusion preventing relationships in G_o in the component graphs is to compute a modified transitive closure on G_o before pulling them apart. A fusion preventing edge are added between two sequential nodes that cannot be placed in the same partition because there exists a path between them that contains at least one parallel node. Similarly, a fusion preventing edge is added between two parallel nodes connected by a path containing a sequential node. We now show how to divide G_o and compute the necessary transitive fusion preventing edges in linear time.

Computing a transitive closure on a DAG is $O(N * E)$ time and space where N is the number of nodes in G_o and E is the number of edges [AHU74]. Of course, transitive closure introduces additional edges that are unnecessary. Applying this algorithm to the graph in Figure 7.2, would result in the following fusion preventing edges: (S_1,S_6), (S_2,S_6), (S_1,S_8), (S_2,S_8), (P_4,P_7). The two of edges, (S_1,S_8) and (S_2,S_8), are redundant because of the original ordering edge (S_6,S_8) and the two other fusion preventing edges (S_1,S_6), (S_2,S_6).

Efficient problem division

Using the following definition, we can determine the minimal number of fusion preventing edges needed between parallel nodes to preserve correctness. Similarly, the minimal number of edges between sequential nodes can be determined.

ALGORITHM 7.4: **Add transitive fusion preventing edges to partition graph**

FindParallelTFPedges (n)
INPUT: $n \in G_o$ a node in the original graph
OUTPUT: G_t partition graph with transitive fusion preventing edges
ALGORITHM:

```
if any predecessor of n is not visited return
mark n visited (reverse postorder walk)
if n ∈Snodes
   Paths(n) = ⋃_{(t,n)∈G_o} Paths(t)
else
   Paths(n) = n
   forall (t, n) ∈ G_o s.t. t ∈Snodes
      forall pnode_i ∈Paths(t)
         addFusionPreventingEdge (pnode_i, n)
      endforall
   endforall
endif
forall (n, m) ∈ G_o FindParallelTFPedges(m)
```

Definition 7.1 Two parallel nodes, $pnode_i$ and $pnode_j$, require a *transitive fusion preventing edge* in their component graph if and only if:

$\forall \; path_k = apathpnode_i \rightarrow snode^+ \rightarrow pnode_j \;\; \in G_o$
1. $\exists \; n \in path_k$ s.t $n \neq pnode_i$ and $n \neq pnode_j$ and
2. $\forall \; n \in path_k, \;\; n \in Snodes$

where $Snodes$ is the set of sequential nodes and $Pnodes$ is the set of parallel nodes in the original graph G_o.

Intuitively, there must exists a path between two parallel nodes, with at least one node on the path and and no parallel nodes are on the path.

Based on this definition, *FindParallelTFPedges* in Algorithm 7.4 computes the necessary transitive fusion preventing edges that must be inserted between parallel nodes. The corresponding algorithm *FindSequentialFTPedges* is specified similarly. *FindParallelTFPedges* formulates this problem like a data-flow problem, accept that solutions along the edges are different depending on the types of nodes an edge connects.

FindParallelTFPedges recursively walks the nodes in reverse postorder, such that a node is never visited until all its predecessors have been visited. For a node n, it computes a set of parallel nodes *Paths*(n), such that *pnode* $\in$ *Paths*(n) if there exists a path from *pnode* to n that contains sequential nodes and does not contain parallel nodes. *Paths*(n) for a sequential node is the union of all n's predecessors *Paths*. *Paths*(n) for a parallel node is itself, n.

If n is a sequential node, no fusion preventing edges need be added. If n is a parallel node, fusion preventing edges are added from each member of *Paths*(t) to n, where t is a sequential node and a predecessor of n.

Performing *FindParallelTFPedges* and *FindSequentialTFPedges* results in a partition graph G_t that includes all the necessary transitive relationships between parallel nodes, and those between sequential nodes. This graph can now easily be separated by placing all the parallel nodes and all the edges between parallel nodes in one graph G_p, and all the sequential nodes and edges between them in G_s. Algorithm 7.5 provides the specifics.

If this algorithm is applied to the example in Figure 7.2 the transitive graph and the component graphs that result appear in Figure 7.3. The fusion preventing edges it adds are (S$_1$,S$_6$), (S$_2$,S$_6$), (P$_4$,P$_7$). Callahan's greedy algorithm may now be applied to the component graphs to obtain a minimal solution for each. The minimal solution for the example places S$_1$ and S$_2$ in the same partition, S$_6$ and S$_8$ in the same partition, P$_3$

ALGORITHM 7.5: **Place sequential and parallel nodes into separate graphs**

PullApart (n)
INPUT: $n \in G_t$ a node in partition graph with transitive fusion preventing edges
OUTPUT: G_p partition graph for parallel nodes
G_s partition graph for sequential nodes
ALGORITHM:
mark n visited
if $n \in$ *Snodes*
add n to G_s
forall (n, m) s.t. $m \in$ *Snodes* add (n, m) to G_s
else $n \in$ *Pnodes*
add n to G_p
forall (n, m) s.t. $m \in$ *Pnodes* add (n, m) to G_p
endif
forall (n, m) s.t. m unvisited **PullApart** (m)

---

and P$_4$ in the same partition, and P$_7$ and P$_5$ in the same partition. This partitioning is illustrated in Figure 7.4 for G_s and G_p.

7.3.2 Merging the solutions

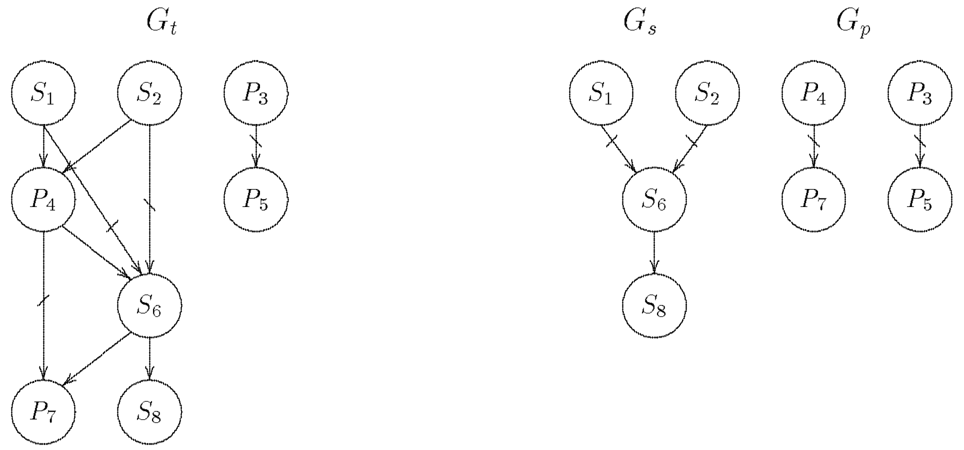
Merging the separate solutions is a straightforward mapping of the edges in G_o to the nodes in G_s and G_p to form a merged graph G_m. The merged graph is formed by placing all the edges and nodes in G_s and G_p into G_m and then adding all the edges in G_o where one endpoint is in G_s and and the other is in G_p. Because the construction of G_s and G_p assures they are both DAGs and G_o is a DAG, G_m will be a DAG. Therefore, it can be topologically sorted into a linear ordering. In Figure 7.4, the merged graph G_m and a linear ordering for our example is presented.

7.3.3 Discussion

The driver for partitioning appears in Algorithm 7.6. It first inserts the required fusion preventing edges. The problem is then divided into two parts and the greedy algorithm performed on each. The resulting solutions are minimal for each part. These solutions are then merged back together to form one overall solution that produces the minimal number of loops achievable without sacrificing parallelism.

These algorithms all take O$(N + E)$ time and space, making them practical for use in a compiler. This algorithmic approach may be applied to other graph partitioning

FIGURE 7.3: **Dividing G_o**



problems as well. The separation of concerns lends itself to other problems that need to sort or partition items of different types while maintaining transitive relationships. In addition, the structure of the algorithm enables different algorithms to be used for partitioning or sorting the component graphs.

The following two sections briefly describe how to use the partitioning algorithm to perform loop fusion and loop distribution.

7.4 Loop fusion

In Algorithm 7.1 *Driver*, candidates for loop fusion are discovered at step (2). The candidate nests have been optimized and may be parallel or sequential loops. In addition, candidates for loop fusion are discovered in Algorithm 7.3 *Order&Parallelize*. Both algorithms need to fuse nests that do not have a common outer loop and may fuse loops that do have a common outer loop. Therefore, fusion of outer distinct loops is attempted first. Fusion is then considered for any candidates created by the outer fusion as well as for candidates present in the original loop structure.

Fusion is always considered last because it may interfere with loop permutation. Permutation is multiplicative and is therefore usually more effective than fusion, which is additive, for creating a larger granularity of parallelism. However, fusion does increase granularity and reduce loop overhead. Fusion also has both good and bad consequences on cache line and register reuse. The merged references may increase data reuse due to completely intersecting references. It may also thwart reuse by causing the data for a single iteration to overflow cache.

FIGURE 7.4: **Fusing G_s & G_p and merging the result**

$\{S_1,S_2\},\{S_6,S_8\},\{P_3,P_4\},\{P_5,P_7\}$

Algorithm 7.6: **Partition Algorithm**

Partition (G_o)
  Input: $G_o = (N, E)$
    N = parallel and sequential loops
    E = data dependence & fusion preventing edges
  Output: $\mathcal{T}$ a parallelizing transformation
  Algorithm:
    forall $n \in G_o$ mark n unvisited
    while ($\exists\ n \in G_o$ marked unvisited) and (n has no predecessors)
      FindParallelTFPedges (n)
    endwhile
    forall $n \in G_o$ mark n unvisited
    while ($\exists\ n \in G_o$ marked unvisited) and (n has no predecessors)
      FindSequentialTFPedges (n)
    endwhile
    forall $n \in G_t$ mark n unvisited
    while ($\exists\ n \in G_t$ marked unvisited) and (n has no predecessors)
      PullApart(n)
    endwhile
    Greedy (G_p)
    Greedy (G_s)
    Merge (G_p, G_s, G_o)

7.5 Loop distribution

In Algorithm 7.3 *Optimize*, loop distribution is considered at step (3) to introduce parallelism when other techniques have failed. Loop distribution seeks parallelism by separating independent parallel and sequential statements in $\mathcal{L}$. If the loop nest contains only a single loop, the partitioning algorithm can be applied to the finest division of the statements in the loop. Although the partitioning algorithm yields maximal theoretical parallelism, it may not be profitable to perform all of the resultant loops in parallel. In this case, all unprofitable parallel loops should be marked sequential and the partitioning algorithm should be applied again.

If there are multiple nests, loop distribution begins by finding the finest division of statements at the outermost loop level. If *Order&Parallelize* can parallelize any of these, then the partitioning algorithm is applied to increase their granularity. For divided loop nests that *Order&Parallelize* cannot parallelize, the outer loop is specified as sequential and this algorithm is applied recursively to the inner nest of loops. Otherwise, the nest is performed sequentially.

7.6 Integrating interprocedural transformations

We now consider integrating the interprocedural loop embedding, loop extraction and procedure inlining to Algorithm 7.1 *Driver*, the driver for parallel code generation. In Chapter 5 we developed additional testing mechanisms in the caller for optimizing nesting structures that cross procedure boundaries. Potential loop and call sequences that may benefit from embedding or extraction are adjacent procedure calls, loops adjacent to calls, and loop nests containing calls. Another candidate for embedding and extraction is a looping structure that contains an outermost loop that *encloses* the body of the called procedure. For example, two adjacent procedure calls may both contain parallel enclosing loops. If these loops may be fused legally and profitably, fusing them is accomplished by first performing loop extraction on both of the procedures. A candidate for procedure inlining in this setting contains loops, but does not contain an enclosing loop.

Candidates for interprocedural optimization are discovered in traversal of the augmented call graph at steps (2) and (3) in Algorithm 7.1. At step (2), independent loops and procedures have been optimized and they are now considered for fusion. As illustrated in Section 5.5, the fusion algorithm is capable of testing and determining fusion of candidate loops and calls. Therefore, no additional mechanisms are required in this case. If a fusion is specified here, loop extraction is the appropriate interprocedural transformation to enable it.

In step (3), a loop structure rooted at m is created. If no attempt at interprocedural transformation is desired, or if there are no calls within the loop structure, m consists of just the loops in the current procedure. Even if no interprocedural transformations are considered, loops containing calls will still be parallelized when profitable. Remember that when interprocedural transformations are considered, they are still only applied if necessary to enable a parallelism enhancing transformation.

If we wish to consider only loop embedding and extraction, then the loop structure for any calls in m that contain enclosing loops is exposed to optimization by including the enclosing loops in $\mathcal{L}$ and ignoring the call site ($\mathcal{L}$ is the loop structure on which *Optimize* is called). The extensions for array sections described in Chapter 5 place annotations at the call and loops for the accessed arrays, allowing them to be treated as normal array references. By using the annotations, the optimization routine is thus permitted to optimize across procedure boundaries when and if necessary. Similarly, if procedure inlining is a desired option, the entire loop structure in a call can be considered at the caller via this method.

Optimize does not specify explicitly the interprocedural transformations that are required to perform the optimizing transformation $\mathcal{T}$ that it returns. The interprocedural transformations are instead implicit in $\mathcal{T}$ given the original program's looping structure.

7.6.1 Selecting the appropriate interprocedural transformation

To apply the set of transformations specified by $\mathcal{T}$, the loops involved may need to be placed in the same routine. In particular, if $\mathcal{T}$ specifies a transformation across a procedure boundary, an interprocedural transformation is required. For example, if $\mathcal{T}$ involves imperfect nesting structures in the caller or the called procedure, then procedure inlining is required to perform $\mathcal{T}$.

If the nesting structures involved in $\mathcal{T}$ are perfect in the caller[5] or are perfect enclosing loops in the called procedure, one of loop embedding or loop extraction is preferred. They are preferable because they do not have the other potential costs of inlining. For example, if there is only one call and its loops are involved in $\mathcal{T}$, then embedding the loop into the called procedure is selected because it reduces procedure call overhead and it does not have inlining's other effects. Additionally, if $\mathcal{T}$ specifies a distribution which results in a single call in a nest, embedding is performed here as well. Otherwise, if there is more than one call involved, extraction is required to place the loops from all the involved calls in the same procedure. Fusion, permutation, etc. may then be performed in the caller.

7.6.2 Extensions to procedure cloning

Interprocedural transformations may induce additional cloning. Remember that given a single level of parallelism, a procedure may be performed sequentially in one calling sequence and in parallel in another. For example, if a procedure is sequential it may be called in a parallel loop or a sequential one. It is correct in either setting. If the caller requires some of its called procedure loops to be parallel, a sequential version

[5] In this case, the perfect loop may contain only calls and no other statements.

minus the extracted loops is needed. Clones are needed if a procedure is called in more than one parallelization setting that require different versions of the procedure. We reuse clones when settings are identical and create them when required. The four potential versions of a procedure are

- a sequential version (only one is required),
- a sequential version that has loops extracted from it (as many versions as different numbers of loops that are extracted for parallelizing distinct callers are required),
- a parallel version (only one is required), and
- a parallel version that has loops embedded into it (as many versions as callers who embed different loops or different numbers of loops are required).

7.7 Discussion

This chapter has presented a general interprocedural algorithm for determining and performing loop-based parallelization. It builds on and extends the algorithms and techniques developed in previous chapters. This algorithm has a few drawbacks; it considers neither multiple levels of parallelism nor important transformations such as loop skewing, loop reversal, and alignment. However, the framework is general enough to support the addition of these types of transformations. The algorithm does perform loop permutation, strip mining, loop fusion, loop parallelization, loop distribution and interprocedural transformations. As we show in Chapter 8, it is very effective in practice.

Chapter 8

Experimental Results

In this chapter we describe an experiment to test the efficacy of the parallel code generation algorithm developed in the previous 5 chapters. A collection of programs hand-coded for parallel machines were obtained for this experiment. From these parallel programs two additional versions were obtained, a *nearby* sequential version, and an automatically parallelized version. The automatically parallelized version was obtained from the nearby sequential version via the parallel code generation algorithm from Chapter 7. Using these program versions we measure the ability of automatic compiler techniques to uncover parallelism that is available in the program. Based on the results of this experiment, we are guardedly optimistic. In many cases, the analysis and algorithms presented in this thesis relieve the programmer of the burden of explicit parallel programming for a variety of shared-memory parallel machines.

8.1 Introduction

A lesson to be learned from vectorization is that programmers rewrote their programs in a vectorizable style based on feedback from their vectorizing compilers [CKK89, Wol89c]. Compilers were then able to take these programs and generate machine-dependent vector code with excellent results. We are testing this same thesis for shared-memory parallel machines. The experiment described below considers the automatic parallelization of sequential program versions where parallelism is known to exist. By measuring the ability of our automatic techniques to uncover this parallelism, we are also testing whether a machine-independent parallel programming style exists. This style would allow compilers to perform machine-specific optimization with excellent results.

We designed the following experiment to measure the efficacy of our automatic parallel code generator. A variety of programs written for parallel machines were assembled. Each of these programs was transformed into a *nearby* sequential version. For each program, a sequential nearby version was easily created by eliminating all the compiler directives. In all the programs, this process resulted in an appropriate machine-independent sequential program version.

On the nearby sequential version, we then simulated by hand our parallel code generation algorithm using the advanced analysis and transformations provided by PFC and the ParaScope Editor PED. We ran and compared all three versions on a Sequent Symmetry S81 with 20 processors. We measured execution times for each version for the entire application, for the portions the user was better able to parallelize, and for the portions our algorithm was better able to parallelize.

8.2 Methodology

8.2.1 Ask and ye shall receive

We solicited programs from scientists at Argonne National Laboratory and from users of the Sequent and Intel iPSC/860 at Rice. The applications programs that were volunteered had been written to run on the following parallel machines: the Sequent Symmetry S81 with 20 processors, the Alliant FX/8 with 16 processors, and the Intel iPSC/860 with 32 processors. The authors are numerical scientists at Rice University, Argonne National Laboratory, ICASE (Institute for Computer Applications in Science and Engineering), George Mason University, Princeton University and the University of Tennessee. All are associated with the Center for Research on Parallel Computation.

The problems inherent to any program test set also arise here. In particular, it may be that only well structured codes were volunteered. Maybe the authors of poorly structured ones were too embarrassed to expose their codes to a critical eye. Fortunately, this furthers our arguments for a modular machine-independent programming style, rather than frustrating us during the experiments. By collecting programs rather than writing them ourselves we avoided the pitfall of writing a test suite to match the abilities of our techniques. No screening process was performed; we used all the programs that were submitted. Table 8.1 contains the name, the abbreviation we use to refer to it, the total number of lines, and the authors of the nine programs in the test suite. They are described in more detail in Appendix A.

8.2.2 Original parallel versions and nearby sequential versions

For each of the programs that were written for the Sequent, this version became the original parallel version. For the programs written for other architectures, any parallelization directives were modified to reflect the equivalent Sequent directives. The nearby sequential versions of each program was created by simply deleting all

TABLE 8.1: **Program Test Suite**

Abbreviation	*Program*	*lines*	*authors*
Interior	Interior Point Method	6153	Guangye Li & Irv Lustig
Direct	Direct Search Methods	1212	Virginia Torczon
Multi	Multidirectional Search Methods	2357	Virginia Torczon
Erlebacher	ADI Integration	1341	Thomas Eidson
Seismic	1-D Seismic Inversion	1712	Michael Lewis
BTN	BTN Unconstrained Optimization	3080	Stephen Nash
Banded	Banded Linear Systems	1834	Stephen Wright
ODE	Two-Point Boundary Problems	3962	Stephen Wright
Control	Optimal Control	2348	Stephen Wright
Linpackd	Linpackd benchmark	772	Jack Dongarra

the parallel directives. In *Erlebacher*, the parallelism was not made explicit. Here, a naive parallelization of outer loops was performed to create the parallel version.

8.2.3 Creating an automatically parallelized version

To create an automatically parallelized program, the nearby sequential program was first imported into the ParaScope Programming Environment [CCH+88, HHK+93]. As a result of importing the program, each procedure in the program was placed in a separate module. Also, a program composition was automatically created that describes the entire program and the call graph was built. At this stage the Program Composition Editor flagged modules that were incorrect. PED then revealed a few minor semantic errors which were corrected. For example, in one program a procedure with many parameters had used the same name twice. Program analysis was also performed automatically.

However, to overcome gaps in the current implemantation of program analysis, we used the Program Composition Editor to import dependence information from PFC. PFC is the Rice system for automatic vectorization (see Section 3.7) [AK87]. PFC's analysis is more mature and includes important features not yet implemented in PED. It performs advanced dependence tests which include symbolics dependence tests and it computes interprocedural constants, interprocedural symbolics and interprocedural mod and ref information for simple array sections [GKT91, HK90, HK91]. PFC produces a file of dependence information that is converted into PED's internal representations.

In PED we used the call graph, program analysis and the transformations that PED provides, to meticulously apply *Driver* (the parallelization algorithm from Chapter 7) to each of the programs by hand. As we discussed in Section 3.3, the implementation of transformation algorithms in PED includes the correctness tests, but does not assist in choosing when or how to apply them. The application of the transformations was completely driven by the *Driver* algorithm. To perform *Driver*, the augmented call graph G_{ac} was easily derived from the call graph. The transformations were attempted as specified by the algorithm, and applied only when PED assured their correctness. Optimization diaries were kept for each program.

8.2.4 Execution environment

For our experiments we used a 20 processor Sequent Symmetry S81 that was provided by the Center for Research on Parallel Computation at Rice University under NSF Cooperative Agreement # CDA8619893. We selected the Sequent for several reasons. The Sequent has a simple parallel architecture which does not include vector hardware, allowing our experiments to focus solely upon medium grain parallelism. Each processor has its own 64Kbyte two-way set-associative cache and the cache line size is 4 words. In addition the Sequent has a very flexible compiler that allows the program to completely specify parallelism and does not introduce all available parallelism [Ost89]. These features gave our algorithms complete control over the parallelization process.

To introduce parallelism into the programs, we used the parallel loop compiler directives suggested by the Sequent's user manual [Ost89]. To compile and run all the program versions, we used the version 2.1 of Sequent's Fortran ATS compiler for multiprocessing with optimization at its highest level (O3). An additional option instructed the compiler to use the Weitek 1167 floating-point accelerator. In a few programs, compiler bugs prevented the highest level of optimization and use of the Weitek chip at the same time. In these programs, the Weitek 1167 floating-point accelerator was used and optimization was suppressed.

We measured execution times for each program version for the entire application, for the portions the user was better able to parallelize, and for the portions our algorithm was better able to parallelize. In programs where the original parallel version and the automatically parallelized versions do not differ, there were no differing portions. For example, if a loop is optimized the same way in both parallel versions, the individual execution time for that loop is not distinguished. However, if the automatic version parallelized a loop and the original did not, the execution time for that loop is measured in all versions. Execution times for the differing optimized portions were measured using the microsecond clock, *getusclk*. The elapsed times for the entire applications were measured in seconds using *secnds*.

8.3 Results

In Table 8.2, we present the speed-up results for the different parallel programs over their sequential counterparts. The results are divided up into three categories with two versions each. The two versions are:

1. *hand* — the original user hand-coded parallel version
2. *auto* — the automatically parallelized version

the three categories are:

1. *Entire Application* — measures the speed-up over the entire application. It also indicates the percent change between the hand-coded and automatic versions.
2. *Degradations* — measures the speed-up in regions where the hand-coded version exploited more parallelism than the automatic version.
3. *Improvements* — measures the speed-up in regions where the automatically parallelized version exploited more parallelism than the original version.

In Table 8.2, a blank entry means that no program or program subpart fell in that category. For example, *Linpackd* did not have an original parallel program version, therefore all the hand-coded slots are left blank. In some cases, differences arose between versions in inner loops. When this situation occurred, the performance of the outer enclosing loop was measured in order to disrupt the execution as little as possible. The speed-ups of these optimized versions are actually under reported. All these programs were complete applications, which read or computed initial data, computed, and printed results. Therefore, linear speed-ups on the entire application were not expected and did not occur.

As can be seen in the percent change column for the entire application category, except for one program, all the automatically generated programs performed the same as the hand-coded parallel version or improved on it. In three programs, *Interior*, *BTN* and *Multi*, the users found more parallelism than our automatic techniques. In *Interior* these degradations did not have much effect on the overall application. If we

TABLE 8.2: **Speed-ups over sequential versions**

	Entire Application			*Degradations*		*Improvements*	
Name	*hand*	*auto*	Δ	*hand*	*auto*	*hand*	*auto*
Seismic	9.1	12.3	35%			3.0	7.9
Erlebacher	13.2	14.2	7%			13.8	15.0
BTN	3.2	4.1	28%	-6.1	1.0	2.0	3.9
Interior	6.9	6.9	0%	6.9	5.2	6.9	10.4
Direct	2.4	2.4	0%				
ODE	3.4	3.4	0%				
Control†	3.8	3.8	0%				
Banded†	$\star$	1.0	$\star$	$\star$	1.0	$\star$	$\star$
Multi	5.3	1.0	-530%	15.1	1.0		
Linpackd		9.2	*NA*				16.5

19 processor Sequent
$\star$: result not obtainable
$\dagger$: 8 processors

look at the table containing the execution times, Table 8.3, it is apparent that both the degradations and improvements only effected a small part of the overall execution time.

In *BTN* and *Multi* the user found parallelism by using critical sections in loops which we were unable to analyze properly. In *BTN*, this parallelism was actually overwhelmed by the overhead of the critical section, resulting in improved performance when executed sequentially. In *Multi*, the parallelism was sufficient to ameliorate the overhead of the critical section, resulting in improved performance for the hand-coded version. In the *Banded* program, the automatic techniques were unsuccessful in finding any parallelism. The reasons for this failure are discussed in Appendix A. Other than these programs, our algorithms either improved performance over the hand coded version or performed equally as well as the hand coded version.

When we consider the improvements category, when our algorithms chose a different optimization strategy from the user, they were always an improvement. This improvement was a least a factor of 1.9 and at best a factor of 4.9.

TABLE 8.3: **Execution Times in seconds**

	Application			*Degradations*			*Improvements*		
	seq	*hand*	*auto*	*seq*	*hand*	*auto*	*seq*	*hand*	*auto*
Seismic	155.97	17.05	12.59				21.14	7.14	2.69
Erlebacher	88.22	6.67	6.20				87.83	6.36	5.86
BTN	44.01	13.93	10.73	0.14	0.85	0.14	13.97	7.045	3.57
Interior	1044.16	151.16	151.53	24.12	3.47	4.64	19.50	2.00	1.87
Direct	151.28	63.65	63.65						
ODE	41.96	12.22	12.22						
Control	17.44	4.61	4.61						
Banded	⋆	⋆	⋆						
Multi	87.60	16.32	87.60						
Linpackd	547.59		59.43				517.87		31.43

⋆ : result not obtainable

8.4 Parallel code generation statistics

Transformations

Besides loop parallelization, the most effective and most often applied code transformation was loop permutation to improve data locality. Outer loop parallelization was also enabled frequently by the memory ordering. In some cases, the loops needed to be strip mined such that memory reuse and parallelization were compatible. For example, loop permutation and strip mining were needed in *Linpackd* and *Erlebacher*. In this experiment, all of the transformation portions of the automatic parallelization algorithm were exercised except for loop distribution and loop embedding. The performance estimator also was used in several instances to inhibit parallelization. An example of these decisions was found in the program *ODE*. Of particular interest in the program *Seismic* were opportunities to perform interprocedural loop extraction and loop fusion which resulted in excellent improvements.

Analysis

The analysis provided by PFC was accurate and for the most part bug free. Some analysis beyond the current implementation was needed to parallelize these programs. Regular section analysis proved to be a very important feature of the current system, but a few improvements are needed. Flow-sensitive summary information about array accesses is need to determine array kills. There were at least two programs that would have benefited from this analysis. In one of these an array could have been made private. Currently, PFC performs symbolic analysis when the symbolic term is a constant. The analysis also needs to perform a more general and sophisticated symbolic test when the symbolic term is unknown and loop invariant. This feature would allow it to better deal with linearized arrays. However, a better solution to this problem, is to reward nicely structured multidimensional array references with excellent performance. Programmers will then have an incentive to program multidimensionally.

Assertions

Five of the programs in this test suite used index arrays that were permutations of the index set [McK90]. Several were monotonic non-decreasing with a well defined with a pattern. In three programs, automatic parallelization would not have been possible with out using an assertion and the testing techniques developed in our earlier research [McK90]. The other two used them in a way that did not interfere with parallelization.

8.5 Discussion

Our results are very promising. They are a clear indication that a clean, modular parallel programming style in Fortran 77 is suitable for portable parallel programming of shared-memory machines given sufficient compiler technology.

Chapter 9

Conclusions

In this research, we undertook to prove an ambitious thesis:

> *Automatic compiler techniques can produce parallel programs with acceptable or excellent performance on shared-memory multiprocessors.*

In this chapter, we summarize what was achieved in pursuit of supporting the thesis. Both the successes and limitations are presented. In addition, the implications of this work for programmers, other architectures, and future work are described.

Due to the limited success of other researchers attacking this problem [EB91, SH91, Sar90], we were not confident when we began that acceptable performance would result from automatic techniques. Therefore, we concentrated much effort on designing and implementing PED. During this process, we developed fast incremental update algorithms for many transformations. These algorithms are useful in both interactive and batch systems because of their speed and precision. Implementing and designing PED also provided insight into the analysis and transformations and a testbed for experimenting with different automatic techniques. Indeed, PED is proving to be a valuable platform for compiling for other parallel architectures as well [HKK+91, DKMC92, HK92], illustrating the usefulness of this type of tool for developing compilers for new types of architectures. At the same time however, we pursued more advanced and general compiler techniques for shared-memory multiprocessors.

We first focused on generalizing existing compiler methods to handle conditional control flow in loops and loop nests that span procedure calls. We developed techniques for loop transformations such as loop fusion when loops contain arbitrary control flow. In particular, the algorithm for performing a given loop distribution in the presence of arbitrary control flow is proven optimal.

We also introduce a new approach which enables optimization across procedure call boundaries without paying the penalties of procedure inlining. Two new transformations, loop embedding and loop extraction, move loops across call boundaries making them available to loop-level optimizations. These transformations are applied judiciously using a goal-directed optimization strategy; the transformations are only applied when they enable performance enhancing optimizations. These interprocedural transformations and the transformations for loops containing conditional control flow provide a general platform for automatic parallelization algorithms for entire applications.

The most significant contribution of this thesis and the core of automatic parallelization is the algorithm that combines introducing parallelism and improving data locality. The algorithm for improving data locality is based on a simple cost model that accurately predicts cache line reuse from multiple accesses to the same memory location and from consecutive memory access. This algorithm is shown effective for uniprocessors as well. Given sufficient granularity, parallelism is then introduced. The algorithm which combines parallelism and data locality uses the cost model to introduce parallelism that complements data locality. This algorithm forms the core of the parallelizing compiler and is shown conclusively to be very effective in practice, illustrating the necessity for considering data locality during parallelization.

The automatic parallelizing compiler further enhances the granularity of parallelism using loop fusion. Also when necessary, the compiler achieves partial parallelism using loop distribution. The loop distribution and loop fusion problems are shown to be duals. A general algorithm for both determines maximal partial parallelism with the minimum number of loops for a collection of candidate sequential and parallel loops. This formulation is shown to maximize parallelism.

Assuming a few assertions that describe index arrays and the range of scalar values, the complete parallel code generation algorithm is then shown effective in practice via experimental results. This result illustrates very promising support of the thesis for shared-memory machines with a local cache and a common bus.

In collecting the test suite, we solicited programs from researchers and then used all the programs we received in our experiment. These programs were carefully hand-coded for good parallel performance and many are currently the best known parallel algorithms. The fact that we were able to improve carefully hand-coded programs designed to exploit parallelism indicates that the details of parallel execution are better left to the compiler.

Many of the parallel loops in the test suite contained procedure calls and control flow. The modular program style that programmers are using to manage complexity proves the need for the generalized parallelization techniques developed in the thesis. In addition, the improvements gained over the hand-coded programs are mostly due to the component algorithm for optimizing data locality in concert with parallelism. Although, loop fusion did prove useful in several programs. We believe our experimental results provide strong evidence for the effectiveness of this approach. With this method, the programmer is permitted to pay more attention to the correctness of a calculation and less to the explicit loop structure required to achieve high performance.

In the test-suite, the vast majority of loops that programmers specified as parallel were able to be detected as parallel by analysis. Most that were not contained unordered critical sections. Programs that use synchronization in order to perform loops in parallel, such as "doacross" style parallelism and critical regions are not handled by our approach [Cyt86, Sub90, HKT92]. However, these loops are an important source of parallelism that should be addressed.

We have not considered the more challenging issues which arise on shared-memory machines with non-uniform access time such as the TC2000 Butterfly. Nor have we considered distributed memory architectures such as the Intel Hypercube. However, other research has shown that many of the same solutions prove effective on both shared-memory and distributed-memory machines [LS90]. For example, the data locality algorithm will be used in a compiler to improve distributed memory performance [HKT92]. Compiling for these architectures is more difficult, but we believe future work will show our techniques to be applicable and that they will serve as a stepping stone in compilation for these machines.

This is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning. Winston Churchill, 10 Nov 1942, after the Battle of Egypt.

Appendix A

Description of Test Suite Programs

Although most of the programs we used are from algorithms that are being used in research and have been published in the literature, they do not come from a single test suite. Therefore, we briefly describe each below. Except for *Linpackd* all the programs were written to run on a parallel machine. We were unable to obtain a parallel version of the *Linpackd* benchmark, but included it anyway because of its importance in the numerical community and its well known algorithms. In the *Implementation details* section for each program, we briefly describe the creation of the different program versions. We also indicate any assumptions or changes that were made to the programs to improve parallelization. As expected, the current analysis was lacking in a few cases. Whenever analysis was required beyond the current implementation, it is noted. Otherwise, all the parallelism detection and transformations were based on the current analysis. The programs are ordered alphabetically.

A.1 Banded Linear Systems

This program is a partitioned Gaussian elimination algorithm with partial pivoting [Wri91a]. The system is assumed to be nonsingular. Hence, the submatrices in the chosen partitioning may be rank-deficient and this makes the algorithm more complex than those which have been proposed for diagonally dominant and symmetric positive-definite systems. It is suitable for multiprocessors with small to moderate numbers of processors. It was written by Stephen Wright at Argonne National Laboratory.

Implementation details

The hand-coded parallel version was written for an Alliant FX/8 using Alliant compiler directives. This version was used for the sequential version. The parallelism consisted of three parallel loops containing a single procedure call each. Using the Sequent directives on those loops does not work. In attempting the automatic parallelization, analysis was complicated by index variables used to perform array linearization based on the program input. With index array assertions and advanced symbolic propagation to differentiate linearized subscripts, dependence analysis will be able to determine independence. However, the program also used offsets into a row at a call site and then subscripted it with negative indices. This practice is not legal Fortran and will thwart even advanced dependence analysis. It is most likely be the bug responsible for the failure of hand-coded and automatic versions.

A.2 BTN Unconstrained Optimization

This program solves unconstrained nonlinear optimization problems based on a block truncated-Newton method [NS91, NS92]. It was written by Stephen Nash and Ariela Sofer at George Mason University. Truncated-Newton methods obtain the search direction by approximately solving the Newton equations via some iterative method. The method used here is a block version of the Lanczos method, which is numerically stable for non-convex problems. This program also uses a parallel derivative checker.

Implementation details

This program was written for execution on the Sequent and therefore required no modifications for parallel execution. The sequential version was easily created by eliminating directives. There were two interesting parallel loops that used a critical section to update a shared variable. Using our analysis and the automatic parallel code generator we were unable to parallelize these loops for this and other reasons. However, as can be seen in the results section, the critical section formed a bottleneck and actually degraded performance beyond that of the sequential performance.

A.3 Direct Search Method

This program is a derivative-free parallel algorithm for the nonlinear unconstrained optimization problem [DT91]. It was written by Virginia Torczon at Rice University. It searches based on the previous function values, where the function is continuous on a compact level set. A special feature of the algorithm embodied in this parallel program is that it is easily modified to to take advantage of any number of processors and to adapt to any ratio of communication cost to function evaluation cost. The parallelism in this version scales with the problem size, but not the number of processors.

Implementation details

This program was written for execution on the Sequent and therefore required no modifications for parallel execution. The sequential version was easily created by eliminating directives.

To automatically parallelize this program required an assertion that an index array used to subscript the data is a permutation array. The four parallel loops could then be identified as such by our tools. Without this assertion, no parallelism could be detected. With the assertion, the critical four loops could be identified as parallel.

The algorithm this program embodies is fully scalable, even though it is not reflected in the results shown earlier. The results are limited because problem size was constrained to 10 to fit on the Sequent. The theoretical speed-up was therefore limited to 10. In addition, the function evaluations available for this study were very small which allowed the overhead of the parallel constructs to become a factor, further degrading performance.

A.4 Erlebacher

Erlebacher is a tri-diagonal solver for the calculation of derivatives written by Thomas Eidson at ICASE, NASA-Langley.

Implementation details

The hand-coded version of Erlebacher provided to us was a sequential version intended for an Intel Hypercube target. We used this as the nearby sequential version and hand performed parallelization on this version. To create the user parallelized version, we performed a naive parallelization of outermost parallel loops.

A.5 Interior Point Method

This program implements a primal-Dual predictor-corrector interior point method to solve multicommodity flow problems [LL92]. It was written by Guangye Li at Rice University and Irv Lustig at Princeton. This problem is a well known application of linear programming. The block structure of the constraint matrix is exploited via parallel computation. The bundling constraints require the Cholesky factorization of a dense matrix, where a method that exploits parallelism for the dense Cholesky factorization is used.

Implementation details

This program was written for execution on the Sequent and therefore required no modifications for parallel execution. The sequential version was easily created by eliminating directives. During the automatic parallelization process, two points of interest were encountered. The first was a small bug, where a parameter was declared twice in a subroutine header that was pointed out by the type checker. The other was that debugging I/O was still present in a procedure called by a parallel loop. Dr. Li indicated the I/O was for development purposes and could be ignored.

A.6 Linpackd Benchmark

The Linpackd benchmark is a representative set of linear algebra routines that are widely used by scientists and engineers to perform numerical operations on matrices [DBMS79]. We used a 200 $\times$200 matrix size for our experiment. This program was written by Jack Dongarra, at the University of Tennessee.

Modifications or assertions

This program was originally a sequential version. From this version, we derived the automatically parallelized version. We performed dead code elimination by hand using constant propagation to delete some special case code for nonunit stride accesses.

A.7 Multidirectional Search Method

The parallel multidirectional search method is a more powerful and general version of the direct search method described above [DT91]. It differs in that the parallelism available in the algorithm is proportional to both the size of the problem and the size of the search space. The search space is based on the number of processors. This algorithm does not just enhance a sequential algorithm, it provides a more ambitious and effective search strategy based on the number of processors and is fully scalable.

Implementation details

This program contained a single parallel loop. Within the loop the programmer used an unordered critical section to test for convergence. This construct could not be analyzed with existing techniques and caused parallelization to fail. More advanced techniques are needed to analyze this programming style.

A.8 1-D Seismic Inversion

This program checks the adjointness of two routines which apply a linear operator DW and its adjoint DW⋆. DW and DW⋆ come from 1-D seismic inversion for oil exploration. The operator DW is the derivative of the incoherency or differential semblance with respect to the background sound velocity. This program was written by Michael Lewis at Rice University.

Implementation details

This program was written for execution on the Sequent and therefore required no modifications for parallel execution. The sequential version was easily created by eliminating directives. The automatically parallelized version of this program employed interprocedural loop fusion to improve performance.

A.9 Optimal Control

This program computed solutions for linear-quadratic optimal control problems that arise from Newton's method or two-metric gradient projection methods to nonlinear problems [Wri91b]. It is a decomposition of the domain of the problem and is related to multiple shooting methods for two-point boundary value problems.

Implementation details

This code was written for an Alliant FX/8 using Alliant compiler directives. This version worked as the sequential version. Using the Sequent parallel loop directives allowed the original parallel version to be obtained. In order to automatically parallelize these loops, array kill analysis and one user assertion about the value of a symbolic were required.

A.10 Two-Point Boundary Problems

This program uses finite differences to solve two-point boundary value Bodes [Wri92]. It was written by Stephen Wright at Argonne National Laboratory. It uses a structured orthogonal factorization technique to solve the system in an effective, stable and parallel manner.

Implementation details

This code was written for an Alliant FX/8 using Alliant compiler directives. This version worked perfectly as the sequential version. The parallelism consisted of three parallel loops containing a single procedure call each. Using the Sequent parallel loop directives allowed the original parallel version to be obtained. In order to automatically parallelize these loops, array kill analysis and better symbolic dependence testing are needed than currently exist in PFC. The symbolic analysis was needed to analyze array linearizations. However, both are well within the scope of an advanced dependence analyzer.

Bibliography

[ABC+87] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the PTRAN analysis system for multiprocessing. In *Proceedings of the First International Conference on Supercomputing*. Springer-Verlag, Athens, Greece, June 1987.

[ABC+88] F. Allen, M. Burke, P. Charles, J. Ferrante, W. Hsieh, and V. Sarkar. A framework for detecting useful parallelism. In *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, July 1988.

[AC72] F. Allen and J. Cocke. A catalogue of optimizing transformations. In J. Rustin, editor, *Design and Optimization of Compilers*. Prentice-Hall, 1972.

[ACK87] 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.

[AHU74] A. V. Aho, J. E. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[AJ90] 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.

[AK84] J. R. Allen and K. Kennedy. Automatic loop interchange. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, Montreal, Canada, June 1984.

[AK87] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[AKPW83] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Conference Record of* *the Tenth Annual ACM Symposium on the Principles of Programming Languages*, Austin, TX, January 1983.

[All83] J. R. Allen. *Dependence Analysis for Subscripted Variables and Its Application to Program Transformations.* PhD thesis, Dept. of Computer Science, Rice University, April 1983.

[All90] J. R. Allen. Private communication, February 1990.

[AS79] W. Abu-Sufah. *Improving the Performance of Virtual Memory Computers.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1979.

[Ban88] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston, MA, 1988.

[Ban90a] U. Banerjee. A theory of loop permutations. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing.* The MIT Press, 1990.

[Ban90b] U. Banerjee. Unimodular transformations of double loops. In *Advances in Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990. The MIT Press.

[BB89] W. Baxter and H. R. Bauer, III. The program dependence graph and vectorization. In *Proceedings of the Sixteenth Annual ACM Symposium on the Principles of Programming Languages*, Austin, TX, January 1989.

[BC86] 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.

[BCHT90] 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.

[BCKT90] M. Burke, K. Cooper, K. Kennedy, and L. Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. Technical Report TR90-126, Dept. of Computer Science, Rice University, July 1990.

[Ber66] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, 15(5):757–763, October 1966.

[BFKK92] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator in the Fortran D programming system. In J. Saltz and

P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*. North-Holland, Amsterdam, The Netherlands, 1992.

[BHMS91] M. Bromley, S. Heller, T. McNerney, and G. Steele, Jr. Fortran at ten gigaflops: The Connection Machine convolution compiler. In *Proceedings of the SIGPLAN '91 Conference on Program Language Design and Implementation*, Toronto, Canada, June 1991.

[BJ66] C. Böhm and G. Jacopini. Flow diagrams, turing machines, and languages with only two formation rules. *Communications of the ACM*, 19(5), May 1966.

[BK89] 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.

[BKK+89] 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.

[Bur90] M. Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Transactions on Programming Languages and Systems*, 12(3):341–395, July 1990.

[Cal87] D. Callahan. *A Global Approach to Detection of Parallelism*. PhD thesis, Dept. of Computer Science, Rice University, March 1987.

[CCH+88] D. Callahan, K. Cooper, R. Hood, K. Kennedy, and L. Torczon. ParaScope: A parallel programming environment. *International Journal of Supercomputing Applications*, 2(4):84–99, Winter 1988.

[CCK88] 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.

[CCK90] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN '90 Conference on Program Language Design and Implementation*, White Plains, NY, June 1990.

[CCKT86] D. Callahan, K. Cooper, K. Kennedy, and L. Torczon. Interprocedural constant propagation. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.

[CDL88] D. Callahan, J. Dongarra, and D. Levine. Vectorizing compilers: A test suite and results. In *Proceedings of Supercomputing '88*, Orlando, FL, November 1988.

[CF87] R. Cytron and J. Ferrante. What's in a name? or the value of renaming for parallelism detection and storage allocation. In *Proceedings of the 1987 International Conference on Parallel Processing*, St. Charles, IL, August 1987.

[CFR+89] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the Sixteenth Annual ACM Symposium on the Principles of Programming Languages*, Austin, TX, January 1989.

[CFS90] R. Cytron, J. Ferrante, and V. Sarkar. Experiences using control dependence in PTRAN. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*. The MIT Press, 1990.

[CHK92] K. Cooper, M. W. Hall, and K. Kennedy. Procedure cloning. In *Proceedings of the 1992 IEEE International Conference on Computer Language*, Oakland, CA, April 1992.

[CHT91] K. Cooper, M. W. Hall, and L. Torczon. An experiment with inline substitution. *Software—Practice and Experience*, 21(6):581–601, June 1991.

[CK87a] D. Callahan and M. Kalem. Control dependences. Supercomputer Software Newsletter 15, Dept. of Computer Science, Rice University, October 1987.

[CK87b] 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.

[CK89] S. Carr and K. Kennedy. Blocking linear algebra codes for memory hierarchies. In *Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing*, Chicago, IL, December 1989.

[CKK89] D. Callahan, K. Kennedy, and U. Kremer. A dynamic study of vectorization in PFC. Technical Report TR89-97, Dept. of Computer Science, Rice University, July 1989.

[CKPK90] 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.

[CKT86a] 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.

[CKT86b] 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.

[CSY90] D. Chen, H. Su, and P. Yew. The impact of synchronization and granularity on parallel systems. In *Proceedings of the 17th International Symposium on Computer Architecture*, Seattle, WA, May 1990.

[Cyt86] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing*, St. Charles, IL, August 1986.

[DBMS79] J. Dongarra, J. Bunch, C. Moler, and G. Stewart. *LINPACK User's Guide*. SIAM Publications, Philadelphia, PA, 1979.

[DCHH88] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. An extended set of Fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, March 1988.

[Die88] H. Dietz. Finding large-grain parallelism in loops with serial control dependences. *Proceedings of the 1988 International Conference on Parallel Processing*, August 1988.

[DKMC92] E. Darnell, K. Kennedy, and J. Mellor-Crummey. Automatic software cache coherence through vectorization. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[DT91] J. E. Dennis, Jr. and V. Torczon. Direct search methods on parallel machines. *SIAM Journal of Optimization*, 1(4):448–474, November 1991.

[EB91] R. Eigenmann and W. Blume. An effectiveness study of parallelizing compiler techniques. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

[FKMW90] K. Fletcher, K. Kennedy, K. S. McKinley, and S. Warren. The ParaScope Editor: User interface goals. Technical Report TR90-113, Dept. of Computer Science, Rice University, May 1990.

[FM85] J. Ferrante and M. Mace. On linearizing parallel code. In *Conference Record of the Twelfth Annual ACM Symposium on the Principles of Programming Languages*, New Orleans, LA, January 1985.

[FMS88] J. Ferrante, M. Mace, and B. Simons. Generating sequential code from parallel code. In *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, July 1988.

[FOW87] 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.

[FST91] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing, Fourth International Workshop*, Santa Clara, CA, August 1991. Springer-Verlag.

[GGGJ88] V. Guarna, D. Gannon, Y. Gaur, and D. Jablonowski. Faust: An environment for programming parallel scientific applications. In *Proceedings of Supercomputing '88*, Orlando, FL, November 1988.

[GJG87] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformations. In *Proceedings of the First International Conference on Supercomputing*. Springer-Verlag, Athens, Greece, June 1987.

[GJG88] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5(5):587–616, October 1988.

[GKT91] 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.

[GP84] A. Goldberg and R. Paige. Stream processing. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 228–234, August 1984.

[Hal91] M. W. Hall. *Managing Interprocedural Optimization*. PhD thesis, Dept. of Computer Science, Rice University, April 1991.

[HHK+93] M. W. Hall, T. Harvey, K. Kennedy, N. McIntosh, K. S. McKinley, J. D. Oldham, M. Paleczny, and G. Roth. Experiences using the ParaScope Editor: an interactive parallel programming tool. In *Proceedings of the*

Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, CA, May 1993.

[HHL90a] 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.

[HHL90b] 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.

[HK90] P. Havlak and K. Kennedy. Experience with interprocedural analysis of array side effects. In *Proceedings of Supercomputing '90*, New York, NY, November 1990.

[HK91] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[HK92] R. v. Hanxleden and K. Kennedy. Relaxing SIMD control flow constraints using loop transformations. In *Proceedings of the SIGPLAN '92 Conference on Program Language Design and Implementation*, San Francisco, CA, June 1992.

[HKK+91] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An overview of the Fortran D programming system. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing, Fourth International Workshop*, Santa Clara, CA, August 1991. Springer-Verlag.

[HKT91] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[HKT92] S. Hiranandani, K. Kennedy, and C. Tseng. Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[HP90] M. Haghighat and C. Polychronopoulos. Symbolic dependence analysis for high-performance parallelizing compilers. In *Advances in Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990. The MIT Press.

[Hus82] C. A. Huson. An inline subroutine expander for Parafrase. Master's thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1982.

[IT88] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the Fifteenth Annual ACM Symposium on the Principles of Programming Languages*, San Diego, CA, January 1988.

[KKLW80a] D. Kuck, R. Kuhn, B. Leasure, and M. J. Wolfe. Analysis and transformation of programs for parallel computation. In *Proceedings of COMPSAC 80, the 4th International Computer Software and Applications Conference*, pages 709–715, Chicago, IL, October 1980.

[KKLW80b] D. Kuck, R. Kuhn, B. Leasure, and M. J. Wolfe. The structure of an advanced retargetable vectorizer. In *Proceedings of COMPSAC 80, the 4th International Computer Software and Applications Conference*, pages 709–715, Chicago, IL, October 1980.

[KKLW84] D. Kuck, R. Kuhn, B. Leasure, and M. J. Wolfe. The structure of an advanced retargetable vectorizer. In *Supercomputers: Design and Applications*, pages 163–178. IEEE Computer Society Press, Silver Spring, MD, 1984.

[KKP+81] 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.

[KM90] K. Kennedy and K. S. McKinley. Loop distribution with arbitrary control flow. In *Proceedings of Supercomputing '90*, New York, NY, November 1990.

[KM92] K. Kennedy and K. S. McKinley. Optimizing for parallelism and data locality. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[KMC72] D. Kuck, Y. Muraoka, and S. Chen. On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup. *IEEE Transactions on Computers*, C-21(12):1293–1310, December 1972.

[KMM91] K. Kennedy, N. McIntosh, and K. S. McKinley. Static performance estimation in a parallelizing compiler. Technical Report TR91-174, Dept. of Computer Science, Rice University, December 1991.

[KMT91a] 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.

[KMT91b] 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.

[KMT92] K. Kennedy, K. S. McKinley, and C. Tseng. Improving data locality. Technical Report TR92-179, Dept. of Computer Science, Rice University, March 1992.

[Knu71] D. Knuth. An empirical study of FORTRAN programs. *Software—Practice and Experience*, 1:105–133, 1971.

[Kuc78] D. Kuck. *The Structure of Computers and Computations, Volume 1.* John Wiley and Sons, New York, NY, 1978.

[KZBG88] U. Kremer, H. Zima, H.-J. Bast, and M. Gerndt. Advanced tools and techniques for automatic parallelization. *Parallel Computing*, 7:387–393, 1988.

[Lam74] L. Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, February 1974.

[Lea90] B. Leasure, editor. *PCF Fortran: Language Definition, version 3.1.* The Parallel Computing Forum, Champaign, IL, August 1990.

[LL92] I. J. Lustig and G. Li. An implementation of a parallel primal-dual interior point method for multicommondity flow problems. Technical Report CRPC-TR92194, Center for Research on Parallel Computation, Rice University, January 1992.

[Lov77] D. Loveman. Program improvement by source-to-source transformations. *Journal of the ACM*, 17(2):121–145, January 1977.

[LRW91] M. Lam, E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, April 1991.

[LS90] C. Lin and L. Snyder. A comparison of programming models for shared memory multiprocessors. In *Proceedings of the 1990 International Conference on Parallel Processing*, St. Charles, IL, August 1990.

[LY88a] 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.

[LY88b] 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.

[McK90] K. S. McKinley. Dependence analysis of arrays subscripted by index arrays. Technical Report TR91-162, Dept. of Computer Science, Rice University, December 1990.

[McM86] F. McMahon. The Livermore Fortran Kernels: A computer test of the numerical performance range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, 1986.

[Mur71] 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.

[NS91] S. G. Nash and A. Sofer. A general-purpose parallel algorithm for unconstrained optimization. *SIAM Journal of Optimization*, 1(4):530–547, November 1991.

[NS92] S. G. Nash and A. Sofer. BTN: software for parallel unconstrained optimization. *ACM TOMS*, 1992. to appear.

[Ost89] A. Osterhaug, editor. *Guide to Parallel Programming on Sequent Computer Systems*. Sequent Technical Publications, San Diego, CA, 1989.

[PGH+90] C. Polychronopoulos, M. Girkar, M. Haghighat, C. Lee, B. Leung, and D. Schouten. The structure of Parafrase-2: An advanced parallelizing compiler for C and Fortran. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*. The MIT Press, 1990.

[Por89] A. Porterfield. *Software Methods for Improvement of Cache Performance*. PhD thesis, Dept. of Computer Science, Rice University, May 1989.

[RC86] B. Ryder and M. Carroll. An incremental algorithm for software analysis. In *Proceedings of the Second ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Palo Alto, CA, December 1986.

[RG89] S. Richardson and M. Ganapathi. Interprocedural optimization: Experimental results. *Software—Practice and Experience*, 19(2), February 1989.

[Ros90] C. Rosene. *Incremental Dependence Analysis*. PhD thesis, Dept. of Computer Science, Rice University, March 1990.

[RP88] B. Ryder and M. Paull. Incremental data flow analysis algorithms. *ACM Transactions on Programming Languages and Systems*, 10(1):1–50, January 1988.

[SA88] K. Smith and W. Appelbe. PAT - an interactive Fortran parallelizing assistant tool. In *Proceedings of the 1988 International Conference on Parallel Processing*, St. Charles, IL, August 1988.

[SA89] K. S. Smith and W. Appelbe. An interactive conversion of sequential to multitasking Fortran. In *Proceedings of the 1989 ACM International Conference on Supercomputing*, Crete, Greece, June 1989.

[Sar90] V. Sarkar. PTRAN — the IBM parallel translation system. In *Proceedings of the International Workshop on Compilers for Parallel Computers*, Paris, France, December 1990. To appear as a chapter in *Parallel Functional Programming Languages and Compilers*, editor B. Szymanski, ACM Press, 1991.

[SAS90] K. Smith, W. Appelbe, and K. Stirewalt. Incremental dependence analysis for interactive parallelization. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[SG90] B. Shei and D. Gannon. SIGMACS: A programmable programming environment. In *Advances in Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990. The MIT Press.

[SH91] J. Singh and J. Hennessy. An empirical investigation of the effectiveness of and limitations of automatic parallelization. In *Proceedings of the International Symposium on Shared Memory Multiprocessors*, Tokyo, Japan, April 1991.

[SK86] R. G. Scarborough and H. G. Kolsky. A vectorizing Fortran compiler. *IBM Journal of Research and Development*, 30(2):163–171, March 1986.

[Sub90] J. Subhlok. *Analysis of Synchronization in a Parallel Programming Environment*. PhD thesis, Dept. of Computer Science, Rice University, August 1990.

[TIF86] 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.

[Tow76] R. A. Towle. *Control and Data Dependence for Program Transformation.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, March 1976.

[WL90] M. E. Wolf and M. Lam. Maximizing parallelism via loop transformations. In *Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990.

[WL91] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Program Language Design and Implementation*, Toronto, Canada, June 1991.

[Wol82] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, October 1982.

[Wol86] M. J. Wolfe. Loop skewing: The wavefront method revisited. *International Journal of Parallel Programming*, 15(4):279–293, August 1986.

[Wol89a] M. J. Wolfe. More iteration space tiling. In *Proceedings of Supercomputing '89*, Reno, NV, November 1989.

[Wol89b] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge, MA, 1989.

[Wol89c] M. J. Wolfe. Semi-automatic domain decomposition. In *Proceedings of the 4th Conference on Hypercube Concurrent Computers and Applications*, Monterey, CA, March 1989.

[Wri91a] S. J. Wright. Parallel algorithms for banded linear systems. *SIAM Journal of Scientific and Statistical Computation*, 12(4):824–842, July 1991.

[Wri91b] S. J. Wright. Partitioned dynamic programming for optimal control. *SIAM Journal of Optimization*, 1(4):620–642, November 1991.

[Wri92] S. J. Wright. Stable parallel algorithms for two-point boundary value problems. *SIAM Journal of Scientific and Statistical Computation*, 1992. to appear.

[Zad84] F. Zadeck. Incremental data flow analysis in a structured program editor. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, Montreal, Canada, June 1984.

[ZBG88] H. Zima, H.-J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.