Loop Transformations for Massive Parallelism

Lee-Chung Lu

YALEU/DCS/RR-937
November 1992

Loop Transformations for Massive Parallelism

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by

Lee-Chung Lu

November 1992

Abstract

Loop Transformations for Massive Parallelism

Lee-Chung Lu
Yale University
1992

Massively parallel technology is the new trend in building supercomputers. Two software challenges must be overcome to make massively parallel machines truly usable: large-scale parallelism detection for fully utilizing the large number of processors, and good data layout for reducing communication overhead. This thesis presents two new classes of loop parallelization techniques to help overcome these difficulties.

The first class of techniques focuses on static methods for scheduling the execution of loops with affine array references, going beyond previous dependence analysis and loop transformation methods by considering dependences under conditionals and generating more general piece-wise affine schedules. The new dependence test takes advantage of special properties of FORTRAN programs, e.g. most coefficients of affine expressions are 1, 0, -1, to obtain accurate dependence information efficiently. The generated piece-wise affine schedules can be different for each disjoint subdomain of the loop iteration space or for each different subset of statements in the loop body. An experiment on the Connection Machine shows that such methods can have dramatic effects on the performance of a transformed program. We also show that the problems studied are tied to existing loop transformation techniques, unified by a theory of loop transformations.

The second class of techniques is for parallelizing loops with indirect array references (FORTRAN style) or pointers (C style). The new technique is a hybrid compiler/run-time approach in which a scheduler is generated by the compiler using static dependence analysis and program slicing. At run-time, the scheduler records the read and write reference patterns and allocates work to processors based on dynamic dependences so deduced. Experiments show that a new optimization technique, called redundant reference elimination, can help to make the run-time scheduling and data layout overhead insignificant for a subclass of loops with indirections or pointers.

Acknowledgement

It is a great pleasure to thank my thesis advisor, Marina Chen, for several years of advice, encouragement, and support. Without her guidance, this thesis would not have been possible. I am especially grateful to her for her untiring attempts to teach me how to formulate and present my work. I would also like to thank the other members of my thesis committee. Ron Cytron offered great encouragement and constructive criticism to my work and did a timely and thorough reading of this thesis despite his busy schedule. Sandeep Bhatt and James Larus both gave valuable comments and advice which have influenced the content and form of the thesis.

I am very grateful to Dennis Shea, Thomas Johnson, and Deborra Zukowski at the IBM T.J. Watson Research Center for spending many hours telling me about the sequential as well as the parallel versions of their circuit simulator. Their work provided me with insights into the systematic approach to parallelizing programs with pointers. My thanks go also to Joel Saltz for helpful discussions on the inspector/executor approach.

I would like to thank Young-il Choo for his valuable suggestions about my writing and speaking. I am very grateful to Chris Hatchell for his careful proofreading of my papers and this thesis. Many individuals in the Crystal Group have influenced my research. I would like to thank Michel Jacquemin, Dong-Yuan Chen, Jim Cowie, Yu Hu, Cheng-Yee Lin, Aloke Majumdar, Joe Rodrigue, Janet Wu, and Allan Yang for many helpful discussions.

It has been an exciting experience belonging to the Yale Computer Science Department. I am grateful to its members for their friendship and inspiration.

Finally, I sincerely thank my wife and our parents for their support and encouragement of my graduate work. The birth of my daughter and my son also gave me a lot of joy.

The generous support provided by an IBM Student Research Associate Fund 1990-91, Office of Naval Research Contract N00014-91-J-1559, and National Science Foundation Contract CCR-8908285 are greatly appreciated.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Massively Parallel Technology

The new trend in building supercomputers with massively parallel technology has posed a great many challenges to software designers.

Traditional supercomputers are built with vector units using exotic circuit technologies that have provided a six-fold speedup for single processor performance over 15 years, from a 160 million floating-point operations per second (MFLOPS) Cray-1 in 1976 to a one billion floating-point operations per second (GFLOPS) single-head Cray Y-MP in 1991. The Cray C90, a 16-head Y-MP that provides 16 GFLOPS and 250 GigaBytes of memory bandwidth, is still the most powerful machine today for a wide range of applications.

However, any further significant increase in single processor performance would have to come at an extremely high cost, as demonstrated by the Cray-3 technology, while the RISC-superscalar architectures based on ever-denser silicon technologies are providing inexpensive, high volume microprocessors at 200 MFLOPS peak performance. The availability of such high-performance commodity microprocessors makes it possible to build a 1024-node Connection Machine CM/5 with 130 GFLOPS peak performance and an aggregate bisection bandwidth of 5 GigaBytes, and a 576-node

Intel Touchstone Delta with 34 GFLOPS peak performance and an aggregate bisection bandwidth of 875 MegaBytes. More processors can be used to build even faster supercomputers. The Intel Paragon can have up to 4096 nodes with 300 GFLOPS peak performance and a 16384-node Connection Machine CM/5 can even reach 2000 GFLOPS peak performance.

In addition to much higher peak performance, massively parallel machines also have much better cost/performance ratios than traditional supercomputers. For example, the cost/performance for the Cray C90 is approximately $2 million per GFLOPS, whereas for the Connection Machine CM/5 it is about $0.2 million per GFLOPS. Therefore, from a hardware point of view, massively parallel machines are much more cost/performance effective than traditional supercomputers.

Unfortunately, the story is different from the software point of view. Because most programs are written in high-level languages, increasing processor performance will speed up original applications while maintaining portability. Only the compilers need to be adapted to new processors, and vectorizing compilers have successfully made traditional supercomputers usable by helping users to vectorize and port many scientific and engineering applications onto these machines. But large numbers of processors in massively parallel machines may also involve significant changes to applications as well as compilers. There are two reasons for this. First, large-scale parallelism, at least insofar as concerns the number of processors, must be exploited to utilize those processors fully. Otherwise, a significant percentage of the processors will be idle and the performance will be far from the potential peak performance of the machines. (For traditional supercomputers, small-scale parallelism in the order of the depth of pipelines is enough to sustain the peak performance.) Second, the communication bandwidth of massively parallel machines is much smaller than the memory bandwidth of traditional supercomputers. Therefore, data layout with high locality to reduce communication overhead becomes a critical issue in the case of massively parallel machines.

The locality issue also exists in memory hierarchy. Compiler optimizations such as those presented in [13,14,32,36,50] can increase the utilization of registers and caches to hide memory latency. These optimizations are useful for both traditional supercomputers and massively parallel machines because they both have memory hierarchy.

Today, traditional supercomputers serve far more different kinds of applications than can massively parallel machines, and the question is whether software can overcome the problems of large-scale parallelism detection and high-locality data layout posed by massively parallel technologies.

1.2 Software Challenges

To make massively parallel machines truly usable, many real applications must be ported. More and more complex and critical applications – e.g. the N-body problem [94,11], circuit simulation [48], data base systems [30] and financial analysis applications [93] – have been found that are very suitable to be parallelized and executed on massively parallel machines in order to gain tremendous performance over traditional supercomputers. However, most of them are written with explicit communication primitives, and this is very tedious and error prone. To make massively parallel machines truly usable, new high-level languages as well as similar kinds of software tools such as vectorizing compilers for traditional supercomputers must be made available. These new high-level languages and parallelizing compilers encounter two new challenges: exploiting large-scale parallelism and optimizing data locality, to gain tremendous performance for applications running on massively parallel machines.

New Languages Recently, several new parallel languages have been developed, notably FORTRAN 90 [37], FORTRAN D [38,45], Vienna FORTRAN [15], CM FORTRAN [22] and ARF [27] (all FORTRAN based), and C* [21], Dataparallel C [43] and DINO [92] (all C based). Most of these new languages have extensions to specify explicit parallelism (e.g. vector notations and `DOALL`) and data layout (e.g.

`DECOMPOSITION`, `DISTRIBUTE` and `ALIGN`).

With these extensions, the burden of parallelism detection and data layout has been transferred from compilers to users. Therefore, these languages are very useful for applications with apparent parallelism using regular (e.g. block and cyclic) data layout that can be reasoned and specified easily by users. However, it is difficult and tedious for users to detect less-apparent parallelism or find a good irregular data layout. Furthermore, if the parallelism is input-data dependent, then vector notations and `DOALL` are not effective in expressing the dynamic parallelism.

A carefully designed parallelizing compiler, on the other hand, is very useful in detecting less-apparent parallelism. We use the following example to show the difficulty in detecting less-apparent parallelism by users. This example is modified and simplified from the dynamic programming example discussed in Chapter 4 in a way that the innermost loop becomes readily vectorizable:

Loop Nest 1.1

```
$INTEGER\ A(1001,1000,1000), B(1001,1000,1000)$
$n = 1000$
DO $(i = n, 1, -1)$ {
  DO $(j = i, n)$ {
    DO $(k = i, n)$ {
      $S_1$ : IF$(i = j)\ A(i,j,k) = B(i+1,n,k)$
      $S_2$ : IF$(i < j)\ A(i,j,k) = A(i+1,j,k)$
      $S_3$ : IF$(2j - i > n)\ B(i,j,k) = B(i,j-1,k) + A(i,j,k)$ } } }
```

The vector form of the above loop is:

Loop Nest 1.2

```
$INTEGER\ A(1001,1000,1000), B(1001,1000,1000)$
$n = 1000$
DO $(i = n, 1, -1)\{$
    DO $(j = i, n)\{$
        $S_1$ : IF$(i = j)\, A(i,j,1:n) = B(i+1,n,1:n)$
        $S_2$ : IF$(i < j)\, A(i,j,1:n) = A(i+1,j,1:n)$
        $S_3$ : IF$(2j - i > n)\, B(i,j,1:n) = B(i,j-1,1:n) + A(i,j,1:n)\ \}\ \}$
```

Although 1000 parallel iterations in the innermost loop of Loop Nest 1.2 can keep the pipelines in vector machines busy, they are not enough for fully utilizing thousands of processors in massively parallel machines. Even using 1000 processors is wasteful because one iteration per processor is too fine-grained and too much communication overhead will be induced. Our new loop transformations presented in Chapter 4 can obtain the following piecewise affine transformation such that two innermost loops of Loop Nest 1.1 become parallelizable:

$$\left\{\begin{array}{l} 2j - i > 1000 \rightarrow (-2i + j, i, k) = (t, i, k) \\ 2j - i \leq 1000 \rightarrow (-i - j + 1000, i, k) = (t, i, k) \end{array}\right\} \tag{1.1}$$

Under this transformation, Loop Nest 1.1 is transformed into the following loop nest:

Loop Nest 1.3

```
$INTEGER\ \ A(1001, 1000, 1000), B(1001, 1000, 1000)$
$n = 1000$
DO $(t = -1000, 998)\,\{$
   DOALL $(i = (1000 - t)/2, \mathsf{max}(1, -t), -1)\,\{$
      DOALL $(k = 1, n)\,\{$
         $j = t + 2i$
         IF$(2j - i > 1000)$
                $S_2$ : IF$(i < j)\, A(i, j, k) = A(i + 1, j, k)$
                $S_3$ : IF$(2j - i > n)\, B(i, j, k) = B(i, j - 1, k) + A(i, j, k)$
         $j = 1000 - t - i$
         IF$(2j - i \leq 1000)$
                $S_1$ : IF$(i = j)\, A(i, j, k) = B(i + 1, n, k)$
                $S_2$ : IF$(i < j)\, A(i, j, k) = A(i + 1, j, k)$                }}}
```

It is very difficult for users simply to verify that the transformation given in Equation 1.1 is valid (i.e. the dependence ordering is preserved). It is even harder for users to figure out the transformation and derive the transformed loop nest.

In addition to programs with less-apparent parallelism, it is also difficult for users to write parallel codes for problems with input-dependent or dynamic-changing structures. Problems with such properties include sparse matrix solvers and partial differential equation solvers using adaptive and unstructured meshes. Consider the following sparse triangular solver given in [98]:

Loop Nest 1.4

```
DO $(i = 1, n)$ {
    DO $(j = low(i), high(i))$ {
        $y(i) = y(i) - a(j) * y(column(j))$ } }
```

Since the values of array *column* are read in during the computation of the program, the parallelism of the loop cannot be determined by the user at coding time or by the compiler. The parallelism of the loop, as well as a good data layout, can only be detected at run time. In this case, vector notations and DOALL are not effective in expressing the dynamic parallelism. And finding a good irregular data layout to reduce communication overhead and achieve load balancing is extra and tedious work for users.

1.3 Novel Compilation Techniques

Having described the software challenges for making massively parallel machines truly usable and the insufficient solutions provided by those new languages, we will now investigate new compiler techniques for massively parallel machines. These new compiler techniques can help to overcome these difficulties for two important classes of programs: those consisting of *affine loops* and those consisting of *nonaffine loops* (to be defined momentarily). And these techniques are not only useful for programs written in sequential languages like FORTRAN and C, but also are applicable to programs written in new parallel languages like FORTRAN 90, if the explicit parallelism is not enough to fully utilize the large number of processors, or the parallelism is input dependent.

Figure 1.1 shows the structure of a parallelizing compiler including the new techniques presented in this thesis and related previous techniques. The dashed-line boxes are the new techniques discussed in this thesis. There are two different compilation

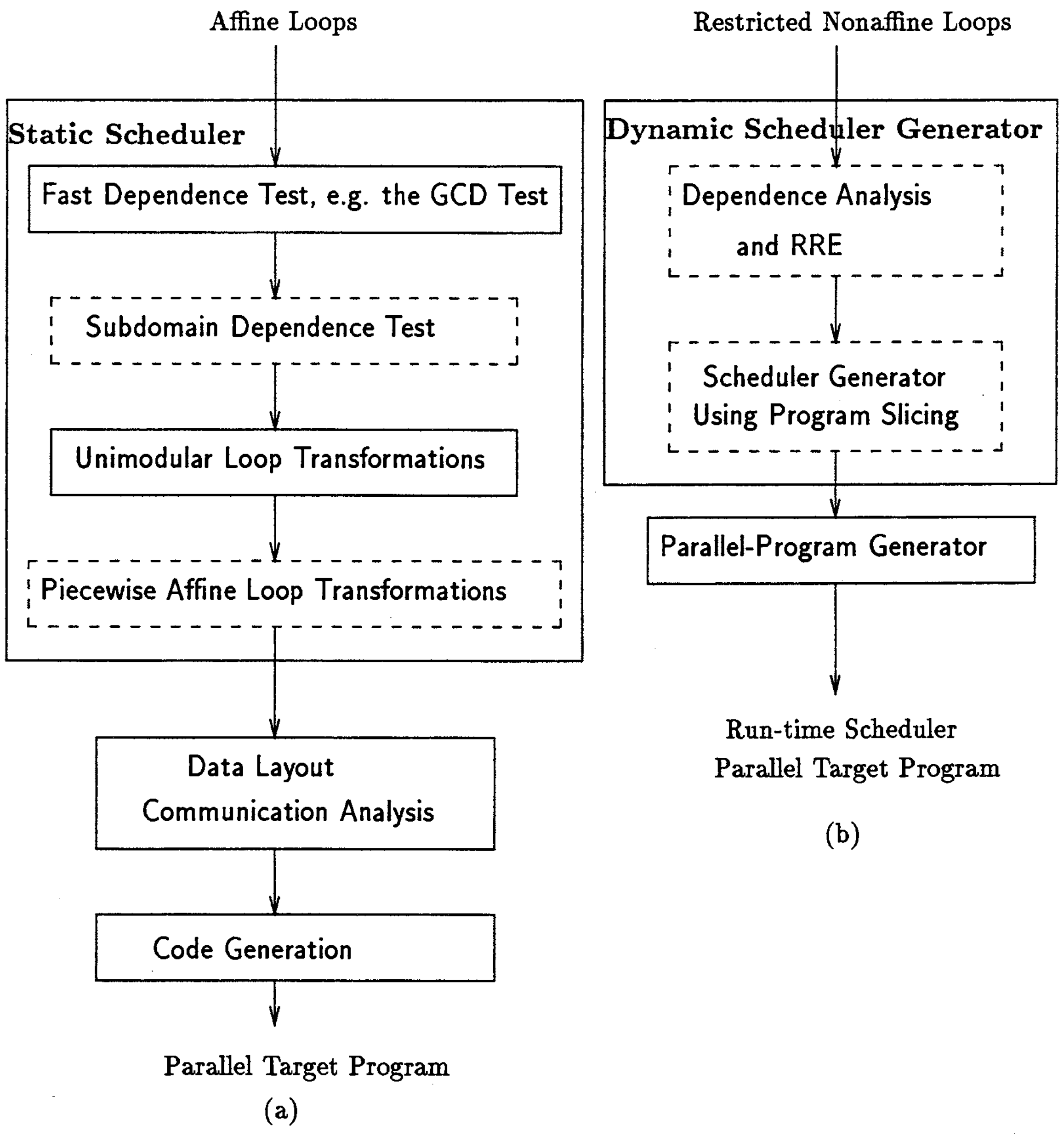

Figure 1.1: The structure of a parallelizing compiler.

processes for two different classes of input programs. Figure 1.1(a) shows a static compilation process for parallelizing *affine loops*, which are loops possibly with conditional statements where the guards as well as the array index expressions are affine expressions of the loop indices. Figure 1.1(b) shows a hybrid compiler/run-time approach for parallelizing *nonaffine loops*, which are loops with indirect array references or pointers. We now give a brief introduction to these two compilation processes.

The compilation process in Figure 1.1(a) consists of three major phases: static parallelism detection, data layout and communication analysis, and code generation. The final outputs of the compiler are target parallel programs. The data layout and communication analysis phase comes into play after the parallelism-detection phase because a good data layout should distribute data which can be computed in parallel to different processors. Data layout, communication analysis and code generation have been extensively studied in [67,68,69,70,71]. Here, we focus on new techniques, i.e. *the subdomain dependence test* and *piecewise affine loop transformations*, to exploit large-scale parallelism.

Parallelizing compilers rely on dependence analysis to detect dependences in application programs. The dependence analysis problem has been studied extensively [3,4,5,6,7,8,12,40,52,54,55,72,74,110,113]. Our subdomain dependence test has two improvements over these techniques. First, the test includes in the system of dependence inequalities the information arising from a program's predicates. Second, it is an improved dependence test for equations with -1, 0, 1 coefficients. The test is more accurate for testing coupled array subscripts in statements with and without conditionals than previous dependence tests.

For loops with true dependences which prevent large-scale parallelization, loop transformation can be applied to reveal parallelism. Our new loop transformations have two improvements. First, we present a complete classification of static loop transformations by viewing the iteration space and statements as distinct dimensions in which code is restructured. From the classification, we have a clear picture of the previous techniques and how they can be extended to detect more parallelism. Second, we provide algorithms to find piecewise affine loop transformations, which can detect more parallelism than *unimodular loop transformations* [10] including loop reversal, interchange, permutation and skewing [4,5,9,58,61,110,111,113]. Experimental results on the Connection Machine CM/2 show that the difference in performance of the transformed codes, which is essentially due to the available parallelism determined by our methods, can amount to two orders of magnitude better than unimodular transformations.

I would like to point out that, although the subdomain dependence test is more accurate for statements with conditionals or with coupled array subscripts, its complexity is higher than some previous dependence tests, e.g. the GCD test [6]. Therefore, the GCD test and other fast dependence tests can be used first to filter out some independent computations. The subdomain dependence test can then be used between the remaining dependent statements to report more accurate dependences. Similarly, piecewise affine loop transformations can detect more parallelism but its complexity is also higher than unimodular loop transformations. Hence, piecewise affine loop transformations are used when unimodular loop transformations cannot reveal enough parallelism.

The compilation process in Figure 1.1(b) shows a hybrid compile-time and run-time approach for parallelizing nonaffine loops, i.e. loops with indirections or pointers. Compile-time techniques like dependence analysis and loop transformations have so far been unsuccessful in parallelizing such loops. Though programs using pointers can be analyzed to some extent [16,42,44,46,49,62,63], those containing input-dependent or dynamic-changing structures are not amenable to compile-time analysis. For such programs, many run-time scheduling techniques have been proposed [26,27,33,80,82,87,95,97,98,101,104,115,114,116]. Clearly, minimizing the run-time scheduling overhead is critical. It is even more critical for massively parallel machines because the scheduling overhead can be significantly amplified by the large-scale parallelism in the source code. We improve the previous techniques by providing new compile-time analysis to make the run-time scheduling overhead insignificant for a subclass of nonaffine loops. We call the loops in this subclass ***restricted nonaffine loops***, and their definition will be given in Section 5.5.1. Two new compiler techniques are developed: (1) *redundant reference elimination* for reducing the run-time scheduling overhead, and (2) *scheduler generation* using program slicing and dependence recording procedures.

1.4 Organization of the Thesis

The thesis is organized as follows.

Chapter 2. Terminologies and definitions of index domain and data dependence used throughout the thesis are given.

Chapter 3. This chapter presents a new data dependence analysis, the *subdomain dependence test*, for conditional statements with coupled array subscripts. Based on the fact that most coefficients of loop indices in linear subscripts and conditionals are either 0, -1 or 1, an efficient algorithm for solving the subdomain dependence test is described. This test is also applicable to statements without conditionals, and to the problem of "dependence cycle breaking" [7].

Chapter 4. This chapter presents new loop transformation techniques that can extract more parallelism than the previous techniques. A formal mathematical framework which unifies the previous loop transformation techniques is given. We classify *schedules* of a loop transformation into three classes: *uniform*, *subdomain-variant*, and *statement-variant*. New algorithms for generating these schedules are given. Viewed from the degree of parallelism to be gained by loop transformation, the schedules can also be classified as *single-sequential level*, *multiple-sequential level*, and *mixed* schedules. We also describe iterative and recursive algorithms to obtain multiple-sequential level and mixed schedules based on the algorithms for single-sequential level schedules.

Chapter 5. This chapter presents a hybrid compiler/run-time approach for parallelizing loops with pointers or indirect array references. A *scheduler* is generated by the compiler based on information deduced from dependence analysis and program slicing. At run-time, the scheduler records dynamic dependences and allocates work to processors based on the run-time reference patterns. New compiler techniques are presented to reduce the run-time scheduling overhead.

Chapter 6. Concluding remarks and directions of future work are presented.

Chapter 2

Definitions and Terminology

Throughout this thesis, programming examples with array references are written in a FORTRAN-like notation, while programming examples with pointers are written in a C-like notation.

2.1 Index Domains and The Generic Loop Nest

Index Domains Let $[a, b]$ be an *interval domain* of integers from a to b. We define an *index domain* D (also called an *iteration space* in [110]) of the following d-level nested loop:

Loop Nest 2.1

```
DO ($i_1 = l_1, u_1$) {
  DO (...) {
    DO ($i_d = l_d, u_d$) {
      loop body }    } }
```

to be the Cartesian product $[l_1, u_1] \times \ldots \times [l_d, u_d]$ of d interval domains $[l_k, u_k]$ for $1 \leq k \leq d$.

For the purpose of formulating loop transformations, we consider D to be a subset of the d-dimensional vector space over rationals. Throughout the thesis, we let

$$\begin{aligned} I &= (i_1, \ldots, i_d), \\ J &= (j_1, \ldots, j_d), \qquad \text{and} \\ K &= (k_1, \ldots, k_d). \end{aligned}$$

With the domain and tuple notations, Loop Nest 2.1 can be rewritten as follows:

Loop Nest 2.2

```
DO ($I$:$D$) {
  loop body }
```

Perfectly and Imperfectly Nested Loops If the statements in the loop body are not interleaved with loop headers defining the loop indices, e.g. DO($i = 1, n$), then we call the loop *perfectly nested.* For example, Loop Nest 2.1 is a perfectly nested loop. If loop bodies are interleaved with loop headers, then we call the loop *imperfectly nested.* For example, Loop Nest 2.3 is an imperfectly nested loop:

Loop Nest 2.3

```
DO ($i = 1, n$) {
  loop body$_1$
  DO ($j = 1, n$) {
    loop body$_2$ }
  DO ($j = 1, n$) {
    loop body$_3$ } }
```

Throughout the thesis, we use the following perfectly nested loop as a generic example for loops with array references, where D is a d-dimensional index domain and $\tau[a]$ is an expression containing a:

Loop Nest L

```
DO ($I$:$D$) {
    ...
    $S_1$ : IF($P_1(I)$) $A(X(I)) = \ldots$
    ...
    $S_2$ : IF($P_2(I)$) $B(Z(I)) = \tau[A(Y(I))]$
    ...
}
```

Figure 2.1: The Generic Loop Nest L.

2.2 Data Dependence

Definitions We now review data dependences between statements. Let S_1 and S_2 be two statements of a program. A *flow dependence* exists from S_1 to S_2 if S_1 writes data that can subsequently be read by S_2 within the same or at a later iteration. An *anti-dependence* exists from S_1 to S_2 if S_1 reads data that S_2 can subsequently overwrite within the same or at a later iteration. An *output dependence* exists from S_1 to S_2 if S_1 writes data that S_2 can subsequently overwrite within the same or at a later iteration. We use the notation $S_1 \Rightarrow S_2$ to denote a dependence from S_1 to S_2.

Lexicographical Ordering We now define relations on elements J and K of a d dimensional index domain. We define "$\prec$" to be the lexicographical ordering: we say $(J \prec K)$ if there exists l, $1 \leq l \leq d$, such that $(j_m = k_m)$ for all m, $m < l$, and $(j_l < k_l)$. We define "$<$" to be an element-wise ordering of "$<$" and say $(J < K)$ if $(j_l < k_l)$ for all l, $1 \leq l \leq d$. Similarly, "$=$" is defined to be $(j_l = k_l)$ for all l, $1 \leq l \leq d$. We say $(J \preceq K)$ if $(J \prec K)$ or $(J = K)$, and $(J \leq K)$ if $(j_l \leq k_l)$ for all l, $1 \leq l \leq d$. We use $\hat{0}$ to denote the zero vector.

Loop-independent and Loop-carried Dependences Consider Loop Nest L. For statement S_2 to compute the value $B(Z(K))$ at iteration K, the value $A(Y(K))$ is needed. If $A(Y(K))$ is computed from statement S_1 at iteration J, i.e. $Y(K) = X(J)$, and $J \preceq K$, then we say S_2 at iteration K is flow dependent on S_1 at iteration J, denoted by $S_1@J \Rightarrow S_2@K$. The same notation is used for anti- and output dependences.

Clearly, if dependence $S_1@J \Rightarrow S_2@K$ exists, then $J \preceq K$ must hold. A dependence $S_1@J \Rightarrow S_2@K$ is either *loop-independent* if $J = K$, or *loop-carried* if $J \prec K$.

Dependence Graph The *dependence graph* for a program segment of n statements consists of n nodes, each node labeled by one statement. For each dependence relation $S_1 \Rightarrow S_2$, there is a corresponding edge in the dependence graph from node S_1 to node S_2.

Equivalence Classes over Statements in a Loop Nest Let "$\overset{*}{\Rightarrow}$" be the reflexive and transitive closure of the dependence relation "$\Rightarrow$" over statements. We define a binary operation "$\sim$" over statements where $S_1 \sim S_2$ if $S_1 \overset{*}{\Rightarrow} S_2$ and $S_2 \overset{*}{\Rightarrow} S_1$. Note that "$\sim$" is an equivalence relation, and therefore, partitions loop statements into equivalence classes (called π blocks in [110]). The technique of *loop fission* [110] can be applied to split the loop nest into several new loop nests, one for each equivalence class.

Dependent versus Independent Blocks We call a statement S *self-dependent* if the dependence relation $S \Rightarrow S$ holds. By the definition of the equivalence relation "$\sim$", a single statement which is not self-dependent can form an equivalence class on its own. This case must be distinguished from all others where cyclic dependences actually occur. We call this special case of an equivalence class under the dependence relation an *independent* block, and others *dependent* blocks (strongly connected components in the dependence graph). For example, consider the following loop nest:

Loop Nest 2.4

```
DO ($i = 1, n$) {
  DO ($j = i + 1, n$) {
    $S_1 : A(i, j - i) = B(i, j - i - 1)$
    $S_2 : B(i, j - i) = A(i - 1, j - i)$
    $S_3 : C(i, j) = A(i, j - i) + B(i - 1, j + 1)$ } }
```

Statements S_1 and S_2 are in the same dependent block because of cyclic flow dependences, and statement S_3 forms an independent block. Loop fission can be applied to split Loop Nest 2.4 into two new loop nests:

Loop Nest 2.5

```
DO ($i = 1, n$) {
  DO ($j = i + 1, n$) {
    $S_1 : A(i, j - i) = B(i, j - i - 1)$
    $S_2 : B(i, j - i) = A(i - 1, j - i)$ } }
DO ($i = 1, n$) {
  DO ($j = i + 1, n$) {
    $S_3 : C(i, j) = A(i, j - i) + B(i - 1, j + 1)$ } }
```

Loop fission can be used to separate an independent block from other dependent blocks, and the new loop nest consisting of one independent block is readily parallelizable. Similarly, loop fission can be used to transform a loop body containing multiple dependent blocks into multiple loop nests, each with a single dependent block.

Clearly, parallelizing one dependent block is easier than parallelizing two blocks at the same time. Therefore, for the purpose of detecting the maximum degree of parallelism, we assume that a loop nest with multiple dependent blocks is separated into multiple loop nests and we only consider loops containing one dependent block from now on. In practice, loop fission will induce smaller granularity in the loop body and extra overheads for repeatedly ranging loop indices over loop boundaries.

Direction Vectors A loop nest consisting of a dependent block may be parallelized by several techniques, namely statement reordering, loop vectorization, interchange and permutation [9,110]. To determine whether these transformations are applicable or not, the notion of a direction vector [110] is necessary.

For a dependence $S_1@J \Rightarrow S_2@K$, the vector $(\mathsf{sig}(k_1 - j_1), \ldots, \mathsf{sig}(k_d - j_d))$ is called a *direction vector* from S_1 to S_2 [110], where sig is a function from the set of integers to the set of ordering relations "<", "=", and ">":

$$\mathsf{sig}(z) = \left\{ \begin{array}{l} z < 0 \rightarrow \text{“<”} \\ z = 0 \rightarrow \text{“=”} \\ z > 0 \rightarrow \text{“>”} \end{array} \right\} \tag{2.1}$$

We use "*" as a shorthand for ("<" or "=" or ">").

Dependence Vectors *Loop skewing* [110,113] is another transformation which may expose more parallelism, if the parallelism gained from the above-mentioned techniques is insufficient. Loop skewing transforms Loop Nest 2.1 as follows: shifting index i_n with respect to index i_m, $1 \leq m < n \leq d$, by a factor of f, where f is a positive integer, replacing l_n with the expression $(l_n + i_m * f)$, replacing u_n with the expression $(u_n + i_m * f)$, and replacing all occurrences of i_n in the loop body with the expression $(i_n - i_m * f)$.

In order to do loop skewing, we need to know the relative positions of index tuples J and K for each dependence $S_1@J \Rightarrow S_2@K$ [83,110]. For a dependence $S_1@J \Rightarrow S_2@K$, the vector $(k_1 - j_1, \ldots, k_d - j_d)$ is called a *dependence vector* from S_1 to S_2 [110].

Since J and K are in the d-dimensional vector space, we use $J+K$ to denote the addition of two vectors J and K, i.e. $J+K=(j_1+k_1,\ldots,j_d+k_d)$; and similarly, $J-K=(j_1-k_1,\ldots,j_d-k_d)$.

Notation for Concatenation Since we will be using matrix and vector notations, we define the notation for matrix concatenation here. We treat a row vector of length d as a degenerate 1-by-d matrix, a column vector of length d as a degenerate d-by-1 matrix, and a scalar as a degenerate 1-by-1 matrix.

A *vertical concatenation* of an m-by-l matrix A and an n-by-l matrix B, denoted by $\begin{bmatrix} A \\ B \end{bmatrix}$, is an $(m+n)$-by-l matrix, where the (i,j)-th element of $\begin{bmatrix} A \\ B \end{bmatrix}$ is equal to the (i,j)-th element of A if $i \leq m$, or it is equal to the $(i-m,j)$-th element of B if $i > m$.

Chapter 3

Subdomain Dependence Test

3.1 Motivation

Parallelizing compilers rely on dependence analysis to detect dependences in application programs. A precise and efficient data dependence analysis is essential to the effectiveness of parallelizing compilers. This dependence analysis problem has been studied extensively [3,4,5,6,7,8,12,40,52,54,55,72,74,110,113]. However, all of the previous dependence decision algorithms are "conservative" in considering data dependence between statements occurring in conditional branches, although some of them are exact if there are no conditionals. By "conservative" we mean that the dependence analysis may report a dependence relation to exist even if there is no dependence due to conditionals. The reasons for not considering an exact dependence analysis for conditional statements in the past have possibly been that (1) the decision algorithm may at the least involve integer programming, which was considered expensive, and (2) the payoff in speedup as a result of a more accurate dependence analysis may not be significant for vector processors or small scale (tens of processors) shared-memory multi-computers. In attempting to parallelize certain algorithms for execution on the Connection Machine, we realize that the potential speedup can be of orders of magnitude because of a more accurate analysis for conditional statements. This chapter

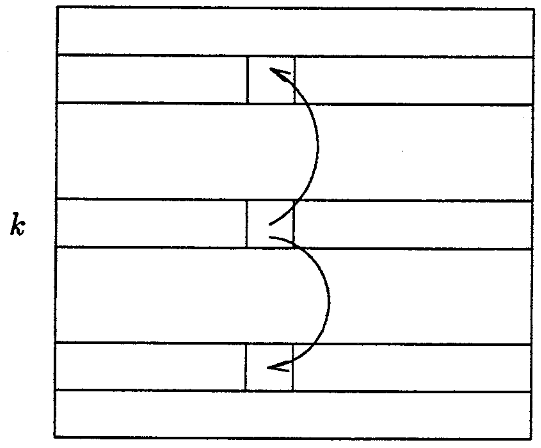

Figure 3.1: Dependences of Gauss-Jordan elimination code.

presents a new dependence analysis for array references in conditional statements. Since expressions of domain indices appearing as guards in conditional statements can be thought of as conditions that divide an index domain into subdomains, we call such a dependence analysis the *subdomain dependence test.*

Let us first take a simple example to show the use of the subdomain dependence test.

An Example: Gauss-Jordan Elimination Consider the following Gauss-Jordan elimination code which reduces the rows of the n-by-n matrix A using the k-th row as pivot:

Loop Nest 3.1

```
DO $(i = 1, n)$ {
  IF$(i \neq k)$ DO $(j = 1, n)$ {
    $A(i,j) = A(i,j) - A(k,j) * DUM(i)$ } }
```

Without considering the conditional $(i \neq k)$, the pivoting row will also be updated and there is dependence between the computation of rows as shown in Figure 3.1.

Therefore, only the inner loop which computes elements of one row can be parallelized. The outer loop, which computes different rows, cannot be parallelized.

However, from the conditional $(i \neq k)$, we know that the pivoting row is not updated in the loop. Therefore, there is no dependence at all and both loops can be parallelized.

Organization of the Chapter In the next section, we formulate precisely the problem of analyzing data dependence between conditional statements. In Section 3.3, we describe the methods and algorithms for solving the subdomain dependence test. Finally, we describe the application of the subdomain dependence test to the problem of "dependence cycle breaking" [7] and the use of the subdomain dependence test in analyzing functional programs.

3.2 Formulating Subdomain Dependence Test

Let us start with formulating dependence analysis without considering conditionals.

3.2.1 Dependence Analysis without Considering Conditionals

Problem Formulation We now formulate dependence analysis for the generic Loop Nest L in Chapter 2 without considering conditionals. For easy reference, Loop Nest L is repeated below:

Loop Nest 3.2

```
DO ($I$:$D$){
    ...
    $S_1$ : IF($P_1(I)$) $A(X(I)) = \dots$
    ...
    $S_2$ : IF($P_2(I)$) $B(Z(I)) = \tau[A(Y(I))]$
    ...                                   }
```

There is a flow dependence from S_1 to S_2 if and only if there exist integer indices I and J satisfying Equations (3.1), (3.2), (3.3) and (3.4). Note that the presence of conditional expressions is ignored:

$$I \in D, \tag{3.1}$$

$$J \in D, \tag{3.2}$$

$$X(I) = Y(J), \quad \text{and} \tag{3.3}$$

$$I \preceq J. \tag{3.4}$$

Equation (3.4) implies that $(d+1)$ direction vectors must be considered. For example, if the dimension d is three, then the direction vectors are $(<, *, *)$, $(=, <, *)$, $(=, =, <)$, and $(=, =, =)$.

Rectangles and Trapezoids Let $\mathcal{Z}^d$ be the d-dimensional Cartesian product of $\mathcal{Z}$. Let l_{ij} and u_{ij} be rational constants where $2 \le i \le d$ and $1 \le j \le d-1$. Let P and Q be two d-tuples of constant rational numbers, and L and U be two constant matrices:

$$L = \begin{pmatrix} 0 & 0 & 0 & \dots & 0 \\ l_{21} & 0 & 0 & \dots & 0 \\ l_{31} & l_{32} & 0 & \dots & 0 \\ & & & \dots & \\ l_{d1} & l_{d2} & l_{d3} & \dots & 0 \end{pmatrix}, \quad U = \begin{pmatrix} 0 & 0 & 0 & \dots & 0 \\ u_{21} & 0 & 0 & \dots & 0 \\ u_{31} & u_{32} & 0 & \dots & 0 \\ & & & \dots & \\ u_{d1} & u_{d2} & u_{d3} & \dots & 0 \end{pmatrix}.$$

A subspace of $\mathcal{Z}^d$ that can be expressed as $\{I \mid (P \leq I \leq Q)\}$ is called a *rectangle* [8]. A subspace of $\mathcal{Z}^d$ that can be expressed as $\{I \mid (LI + P \leq I \leq UI + Q)\}$ is called a *trapezoid* [8].

Traditional dependence analysis [8] restricts that an index domain of a do loop be either a rectangle or a trapezoid, or a union of such domains. For the purpose of a dependence analysis, each member domain of the union can be considered separately. Hence we can further restrict, without loss of generality, that an index domain of a do loop be a rectangle or a trapezoid. Conventional dependence analysis also restricts all subscript functions, e.g. X, Y and Z in Loop Nest 3.2, to be affine functions of the loop indices.

The problem of previous dependence analysis is to determine if there exist I and J that satisfy Equations (3.1), (3.2), (3.3) and (3.4), where conditionals are not considered.

Previous Algorithms Typical methods for analyzing include the *single index exact test* [7,8,110,113] for single-level loop, which can be efficiently performed to analyze indices which are integers.

For one-dimensional arrays in a multi-level loop, methods for testing integer indices include those described in Section 3.3 in [7] and Section 2.5.4 in [113]. Both algorithms have the disadvantage that they require time exponential in the number of loop levels. There are several efficient algorithms for testing rational indices for one-dimensional array references [3,4,5,6,7,8,72,110,113].

For multi-dimensional arrays in a multi-level loop, integer programming [65,99] is needed for testing integer indices. A faster algorithm is the λ test [74], which determines if there exist rational indices for multi-dimensional array references.

Other methods include using the theories of Diophantine equations [8,72] to decide whether Equation (3.3) has an integer solution. The GCD test [6] is used to decide if Equation (3.3) has *integer* solutions when the number of equality in Equation (3.3) is one [8]. The I test [52,54,55] improved the accuracy of the GCD test for one-dimensional array references.

The efficiency of some of these dependence decision algorithms is partly due to the fact that the index domains of loops, i.e. domain D in Equations (3.1) and (3.2), are either rectangles or trapezoids. In the next section, we will show that this fact is no longer true when conditionals are considered.

3.2.2 Subdomain Dependence Test

We will now formulate the same problem with conditionals taken into consideration.

Notation and Definitions Let h_k and c be rational constants where $1 \leq k \leq d$. Let H be a d-tuple $(h_1, h_2, \ldots, h_d)$. A subspace of $\mathcal{Z}^d$ that can be expressed as $\{I \mid HI = c\}$ is called a *hyperplane*. A subspace of $\mathcal{Z}^d$ that can be expressed as $\{I \mid HI \leq c\}$, is called a *half space*. The intersection of a finite number of hyperplanes and half spaces is called a *polyhedron*. A bounded polyhedron is called a *polytope* [100].

A conditional expression is said to be in *disjunctive normal form* if it is the disjunction of conjunctions of predicates. It is well-known that any conditional expression can be transformed into disjunctive normal form.

Consider a statement S with a conditional expression P in a loop over index domain D:

```
DO (I:D){
    ...
    S : IF(P(I)) ...
    ...              }
```

We define the index domain of statement S to be D under the restriction of P, denoted by $(D{\downarrow}P)$:

$$D{\downarrow}P = \{I \mid I \in D, \text{and } P(I) \text{ is true}\}. \tag{3.5}$$

Problem Formulation Now consider the generic Loop Nest 3.2. A subdomain dependence test is formulated as: A flow dependence from S_1 to S_2 exists if and only if there exist integer index tuples I and J satisfying

$$I \in (D{\downarrow}P_1), \tag{3.6}$$
$$J \in (D{\downarrow}P_2), \tag{3.7}$$
$$X(I) = Y(J), \quad \text{and} \tag{3.8}$$
$$I \preceq J. \tag{3.9}$$

We restrict the form of a conditional expression to be an affine expression of the indices. Under this restriction, a predicate in the disjunctive normal form of a conditional expression is either an equality $(HI = c)$ or an inequality $(HI \leq c)$. Therefore, a conjunction of predicates specifies a polyhedron and a disjunction specifies a union of polyhedra. Since we can test multiple polyhedra one by one, it suffices to consider predicates that specify a polyhedron. To summarize, the index domain of a statement in a conditional branch generated by predicate P is $(D{\downarrow}P)$, an intersection of a rectangle or a trapezoid with a polyhedron, which is, in general, a polytope.

Note that a rectangle or a trapezoid is a polytope, but not vice versa. This is another way of seeing that, in the presence of conditional expressions, a dependence decision algorithm is conservative if it can only be applied to a rectangle or trapezoid that encloses the polytope in question.

Since Equations (3.6),(3.7) and (3.8) specify a polytope while Equation (3.9) requires the consideration of $(d+1)$ direction vectors, the union of $(d+1)$ polytopes must be considered. A flow dependence from statement S_1 to statement S_2 exists in Loop Nest 3.2 if and only if the $(d+1)$ polytopes contain integer points. It appears that to obtain both flow and anti-dependence information, $(2d+1)$ polytopes must be tested for each pair of statements S_1 and S_2 in Loop Nest 3.2. This complexity can be reduced by applying the techniques of the ***hierarchical dependence test*** [12] so as to reduce the number of direction vectors that need to be examined and thus the number of polytopes to be tested.

How Hard is Subdomain Dependence Test? If a polytope cannot be characterized as a member of a special class, e.g. rectangles or trapezoids, then we say it is a *general* polytope.

The following theorem states that the complexity of a subdomain dependence test is high.

Theorem 3.1 A subdomain dependence test is as hard as deciding whether a general polytope contains an integer point.

Proof.

Consider a general polytope T specified by $T = \{I \mid AI \leq B\}$. We need to prove that deciding if T contains an integer point is polynomial-time reducible to a subdomain dependence testing problem. Consider the following loop nest:

Loop Nest 3.3

```
DO $(I{:}D)\{$
    $S_1$ : IF$(P_1(I))\, A(I) = \ldots$
    $S_2$ : IF$(P_2(I))\, \ldots = A(I)\ \}$
```

The problem of the subdomain dependence test between statements S_1 and S_2 is formulated as: A flow dependence from S_1 to S_2 exists if and only if there exist integer index tuples I and J satisfying $(I \in (D{\downarrow}P_1))$, $(J \in (D{\downarrow}P_2))$, and $(I = J)$. It is easy to see that the dependence exists if and only if domain $(D{\downarrow}P)$ contains an integer point, where $P(I)$ is the conjunction of $P_1(I)$ and $P_2(I)$. We can make $P(I)$ to be the conjunction of the inequalities in $AI \leq B$ so that $(D{\downarrow}P) = (D \cap T)$. The key point is to find an index domain D in polynomial time such that $(D \cap T)$ contains an integer point whenever T contains an integer point. If this can be achieved, then deciding if T contains an integer point is polynomial-time reducible to deciding if $(D{\downarrow}P)$ contains an integer point, which is a subdomain dependence testing problem. A theorem by Schrijver (Corollary 17.1b in [100]) said that: If $AI \leq B$ has an integer solution, then it has one of *size* polynomially bounded by the sum of the sizes of A and B. (For a matrix $A = (a_{ij})_{1 \leq i \leq c, 1 \leq j \leq d}$, size$(A) = c * d + \sum_{ij}(1 + \lceil \log_2(|a_{ij}| + 1) \rceil)$.) Let s be the sum of the sizes of A and B. We can construct the index domain D to be $\{(i_1, \ldots, i_d) \mid -2^s \leq i_l \leq 2^s, 1 \leq l \leq d\}$ so that D is large enough to contain an integer point in T if one does exist. □

A theorem by by Schrijver (Theorem 18.1 in [100]) said that: Given matrix A and vector B, the problem for deciding whether $AI \leq B$ has an integer solution is NP-complete. By Theorem 3.1 and Schrijver's theorem. we know that the subdomain dependence testing problem is NP-complete.

I would like to point out that there is still no literature discussing the complexity of the exact dependence analysis for array references with coupled subscripts in statements without conditionals. In proving the subdomain dependence test problem to be NP-complete, we use the property that the conditionals can contain *general* linear predicates of loop indices. Without conditionals, this generality will be replaced by the confined property of index domains: an index domain of a loop is either a rectangle, or a trapezoid, or a union of such domains. Therefore, the proof of Theorem 3.1 cannot be used to prove the complexity of exact dependence analysis between statements without conditionals.

3.3 Algorithms for the Subdomain Dependence Test

Recently, an empirical study of program characteristics [102] found that 93% of the coefficients of loop indices in linear subscript functions are either 0, -1 or 1. We will assume that most coefficients of loop indices in linear conditional expressions are either 0, -1 or 1. By taking advantage of 0, -1 and 1 coefficients, we present an efficient solution for this special case in Section 3.3.1. We then give a general solution for the subdomain dependence test in Section 3.3.2, and discuss unknown loop bounds in Section 3.3.3.

3.3.1 Solution for the Special Case

The basic idea of our solution for the special case with 0, -1, or 1 coefficients is as follows. Let T be the polytope to be tested. We can use linear programming to find whether T contains any real points. If not, then there is no integer point in T and we are done. Otherwise, let R be an arbitrary real point in T. We will show that if T contains any integer points at all, then it contains an integer point very close to R. Therefore, to determine if there are any integer points in T, it suffices to check for integer points close to R.

Comparing with Banerjee's and Li's Methods Banerjee [6] and Li [72] have developed efficient dependence decision algorithms for this special case with further restrictions. Banerjee [6] found that if the index domain of the loop is a rectangle and no conditionals are considered, and arrays are one-dimensional, then the test for real indices is also valid for integer indices. Li [72] extended Banerjee's result to two-dimensional array references.

Our approach does not restrict the index domain to be a rectangle or trapezoid. Therefore, it is useful for statements with and without conditionals. Further, our method has no restrictions on the dimensions of arrays, and is more efficient for lower dimensional polytopes. However, the generality of our result does not come for free. Our approach requires linear programming to find the coordinates of a real point in the polytope, while Banerjee's and Li's methods only need to test if there is a real point.

The following problem remains open: Is there a fast algorithm for Banerjee's and Li's methods for multi-dimensional arrays when conditionals are not considered?

The Solution We now describe the solution for the special case. Let T be the polytope to be tested. Clearly, T can be expressed as $T = \{I \mid AI \leq B\}$, where A is an integer matrix and B is a vector. Theorem 3.2 states that the difference between any real point R and its closest integer point I in T, i.e. $R - I$, depends on a set of vectors, called a *basis*, which is determined by A. We then tighten the relation between R and I by constructing the basis explicitly. Lemmas 3.3 and 3.4, together with a theorem given by Fernandez and Quinton [35], provide a precise definition of the basis, from which we know how the basis can be constructed systematically. The relation of R and I in terms of the vectors in the basis is given in Lemma 3.5, which describes a systematic way to find all integer points I that need to be tested for multi-dimensional polytopes in general. Theorem 3.6 gives the number of such integer points for the special cases when all elements in A are either 0, -1 or 1 especially for 2, 3 and 4-dimensional polytopes.

Below, we use $\|V\|$ to denote the length of a vector V, and use $\Delta(A)$ to denote the maximum of the absolute values of the determinants of the square submatrices of A. Theorem 3.2 is very similar to a theorem by Cook *et al.* [23], which gives a bound for the distance between any optimal solution to a linear program and the nearest optimal integer solution.

Theorem 3.2 If T contains integer points, then for any real point R in T there exists an integer point I in T with $\|R - I\| < t\Delta(A)$, where t is the dimension of T.

Proof. Assume Z is an integer point in T. Split A into submatrices A_1 and A_2 such that $A_1R \geq A_1Z$ and $A_2R < A_2Z$. Let C be a *cone* $C = \{X \mid A_1X \geq 0, A_2X \leq 0\}$. (A nonempty set C of points in Euclidean space is called a *cone* if $\lambda x + \mu y \in C$ whenever $x, y \in C$ and $\lambda, \mu \geq 0$.) Let $\mathcal{G}$ be a finite set of integer vectors which generates C (so $\mathcal{G}$ is a basis of C and any vector in C can be written as a nonnegative linear combination of c vectors in $\mathcal{G}$, where c is the dimension of C, $c \leq t$ and c is less than or equal to the number of vectors in $\mathcal{G}$). Using Cramer's rule we may assume that the length of each vector in $\mathcal{G}$ is less than or equal to $\Delta(A)$. Since $R - Z \in C$, there exist numbers $\lambda_i \geq 0$ and vectors $G_i \in \mathcal{G}$, $1 \leq i \leq c$, such that $R - Z = \lambda_1 G_1 + \ldots + \lambda_c G_c$. Let

$$\begin{aligned} I &= Z + \lfloor\lambda_1\rfloor G_1 + \ldots + \lfloor\lambda_c\rfloor G_c \\ &= R - (\lambda_1 - \lfloor\lambda_1\rfloor)G_1 - \ldots - (\lambda_c - \lfloor\lambda_c\rfloor)G_c. \end{aligned} \tag{3.10}$$

Since Z is integral and $G_1, \ldots, G_c$ are integral, I is also integral. Furthermore,

$$\begin{aligned} A_2I &= A_2Z + \lfloor\lambda_1\rfloor A_2G_1 + \ldots + \lfloor\lambda_c\rfloor A_2G_c \leq A_2Z \\ \text{and} \quad A_1I &= A_1R - (\lambda_1 - \lfloor\lambda_1\rfloor)A_1G_1 - \ldots - (\lambda_c - \lfloor\lambda_c\rfloor)A_1G_c \leq A_1R. \end{aligned}$$

So $AI \leq B$ and I is an integer point in T. Finally,

$$\begin{aligned} \|R - I\| &= \|(\lambda_1 - \lfloor\lambda_1\rfloor)G_1 + \ldots + (\lambda_c - \lfloor\lambda_c\rfloor)G_c\| \\ &< \|G_1\| + \ldots + \|G_c\|. \end{aligned}$$

So $\|R - I\| < c\triangle(A) \leq t\triangle(A)$. □

As described in the proof of Theorem 3.2, the length of each vector in $\mathcal{G}$ is less than or equal to $\triangle(A)$. We now sharpen the bounds on $R - I$ by constructing $\mathcal{G}$ explicitly. Let $A' = \begin{pmatrix} -A_1 \\ A_2 \end{pmatrix}$, then C can also be expressed as $C = \{X \mid A'X \leq 0\}$. Fernandez and Quinton [35] showed that $\mathcal{G}$ does not contain any redundant vectors, i.e. any vector in $\mathcal{G}$ cannot be represented as a positive linear combination of other vectors in $\mathcal{G}$, if and only if for all distinct vectors G and G' in $\mathcal{G}$, there is a row V of A' such that $VG = 0$ and $VG' < 0$. We can also assume that $\mathcal{G}$ contains no zero vectors and let each $G = (g_1, g_2, \ldots)$ in $\mathcal{G}$ be normalized in the sense that the greatest common divisor of g_1, g_2, ... is 1. From Fernandez and Quinton's theorem we have the following lemma:

Lemma 3.3 A vector G in $\mathcal{G}$ is not redundant if and only if there does not exist another vector G' in $\mathcal{G}$ such that for every row V of A', $VG = 0$ implies $VG' = 0$.

Let $\mathcal{A}(t, n)$ be a matrix whose rows contain all possible vectors of length t and with elements being in the interval $(-n, n)$. For example, the 9 rows of $\mathcal{A}(2, 1)$ are $(0, 0)$, $(0, \pm1)$, $(\pm1, 0)$ and $(\pm1, \pm1)$ (we use $(\pm1, \pm1)$ as a shorthand for $(1, 1)$, $(1, -1)$, $(-1, 1)$ and $(-1, -1)$). Recall that the length of each vector G in $\mathcal{G}$ is less than or equal to $\triangle(A)$ as described in the proof of Theorem 3.2. Together with Lemma 3.3, we have the following lemma:

Lemma 3.4 If A has t columns and all elements in A are in the interval $(-n, n)$, then all possible vectors in $\mathcal{G}$ are in the following set $\mathcal{B}(t,n)$:

$$\begin{aligned}\mathcal{B}(t,n) = \{G \mid \ & \|G\| \leq \Delta(\mathcal{A}(t,n)), \text{ and} \\ & \text{there does not exist another vector } G' \text{ in } \mathcal{B}(t,n) \\ & \text{such that for every row } V \text{ of } \mathcal{A}(t,n), \\ & VG = 0 \text{ implies } VG' = 0\}.\end{aligned}$$

Lemma 3.4 gives a systematic way to generate $\mathcal{B}(t,n)$. Two examples of $\mathcal{B}(t,n)$ are:

$$\begin{aligned}\mathcal{B}(2,1) &= \{\ (0,\pm1),(\pm1,0),(\pm1,\pm1)\}, \\ \mathcal{B}(3,1) &= \{\ (0,0,\pm1),(0,\pm1,0),(\pm1,0,0),(0,\pm1,\pm1),(\pm1,0,\pm1), \\ &\qquad (\pm1,\pm1,0),(\pm1,\pm1,\pm1),(\pm1,\pm1,\pm2),(\pm1,\pm2,\pm1),(\pm2,\pm1,\pm1)\}.\end{aligned}$$

Let R be a real point and I be an integer point. Clearly, R can be represented as a sum of an integer point R_I and a real point R_R, i.e. $R = R_I + R_R$, where $(0,\ldots,0) \leq R_R < (1,\ldots,1)$. The following lemma relates I to R_I:

Lemma 3.5 If R and I satisfy Equation (3.10), then there exist numbers β_i, $0 \leq \beta_i < 1$, and vectors G_i in $\mathcal{B}(t,n)$, $1 \leq i \leq t$, such that

$$(0,\ldots,0) \leq I - R_I + \beta_1 G_1 + \ldots + \beta_t G_t < (1,\ldots,1). \tag{3.11}$$

Proof. From Equation (3.10), we can let $\beta_i = \lambda_i - \lfloor \lambda_i \rfloor$ for $1 \leq i \leq c$ and let $\beta_i = 0$ for $c < i \leq t$. Therefore, $0 \leq \beta_i < 1$ holds for $1 \leq i \leq t$ and

$$R - I = \beta_1 G_1 + \ldots + \beta_t G_t.$$

Replacing R by $R_I + R_R$, we have

$$R_R = I - R_I + \beta_1 G_1 + \ldots + \beta_t G_t.$$

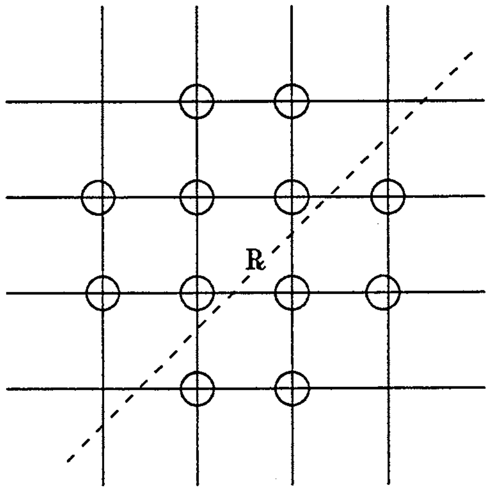

Figure 3.2: Twelve integer points to be tested for a 2-dimensional polytope.

Since $(0,\ldots,0) \leq R_R < (1,\ldots,1)$, we have the proof. □

We can write a program to find all integer points I surrounding an integer point R_I such that Equation (3.11) holds. If all elements of A are either 0, -1 or 1, then we find that there are 12 such I when T is 2-dimensional, 776 such I when T is 3-dimensional and 708480 such I when T is 4-dimensional. We thus have the following theorem:

Theorem 3.6 Let all elements of A be either 0, -1 or 1. Let R be a real point in T. We need test no more than 12 integer points surrounding R to decide whether T contains integer points when T is 2-dimensional, 776 points when T is 3-dimensional. and 708480 points when T is 4-dimensional.

For example, Figure 3.2 shows the 12 integer points surrounding R to be tested for 2-dimensional polytopes.

Let $aI \leq b$ be an inequality in $AI \leq B$ which specifies the polytope to be tested. If $aR = b$ holds, i.e. the real point R is on the hyperplane $\{I \mid aI = b\}$, then the half space $\{I \mid aI \leq b\}$ contains roughly half of those integer points given in Theorem 3.6. The other half will be ruled out by $aI \leq b$. For instance, in Figure 3.2, 7 points are on one side of the dotted line and 5 points are on the other side. Therefore, we can first find an inequality $aI \leq b$ in $AI \leq B$ such that $aR = b$ holds, we then only need to test 6, 388 and 354240 integer points in average for 2, 3 and 4-dimensional polytopes respectively.

3.3.2 General Solution

We now discuss the solution of the subdomain dependence test when the polytope T to be tested does not fall under the above special case.

If the condition of 0, -1, 1 coefficients holds but the dimension of T is larger than 4, then we can try to reduce the dimension of T by the following two methods:
(1) *Linearly Independent Equalities:* Assume T is specified by n linearly independent equalities. We can use the methods given in Chapter 5 in [8] to solve the Diophantine equations of these n equalities to decrease the dimension of T by n.
(2) *Subsystems:* For any two variables appearing in the same equalities or inequalities, we say they are in the same *equivalence class.* If there are more than one equivalence class of the variables (de-coupled indices), then the system of equalities and inequalities can be separated into subsystems according to the equivalence classes of variables. Clearly, any empty subsystem implies that the original system is also empty.

If the dimension of T cannot be reduced to be less than or equal to 4, or the system contains coefficients other than 0, -1 or 1, then integer programming is used to find integer points. Feit's algorithm [34] can solve two-dimensional integer programming problems in $O(n \log n)$ time where n is the number of constraints. And Scarf's *basis reduction* algorithm [99], a general integer programming solver, can test any polytope for integer points.

3.3.3 Unknown Loop Bounds

At compile time, some loop bounds may be specified by unknown constants. Clearly, techniques such as constant propagation should be applied first. For the remaining unknown constants, we will treat them as extra variables. Therefore, a t-dimensional polytope T specified by u unknown constants can be considered as a $(t+u)$-dimensional index domain T' without unknown constants. Since there are no bounds for these extra variables, T' can be a polyhedron. Theorem 3.6 and Feit's [34] algorithm are also applicable to polyhedra. The basis reduction algorithm [99] cannot test if a polyhedron is free of integer points but it can determine whether T' is a polytope or a polyhedron. If Theorem 3.6 and Feit's [34] algorithm cannot be applied and T' is a polyhedron containing real points, then we assume that T' contains integer points, yielding a more conservative result.

3.3.4 Summary of Subdomain Dependence Testing Steps

To summarize, the subdomain dependence test consists of the following steps:

1. Formulate the systems of dependence inequalities according to Equations (3.6), (3.7), (3.8) and (3.9) and the hierarchical dependence test [12]. (There can be as many as $(2d+1)$ systems, where d is the dimension of the loop nest.)

2. For each system, repeat the following steps:

3. Treat unknown loop bounds as extra variables, which increase the dimension of the system.

4. Reduce the dimension of the system by solving linearly independent equalities and finding subsystems of equalities and inequalities.

5. For each subsystem with 0, -1 and 1 coefficients and 2, 3 or 4 dimension, do steps 6, 7 and 8. Otherwise, do step 9.

6. Use linear programming to find a real point R in the subsystem.
7. If there is no such R, then the original system is empty and no dependence exists.
8. Otherwise, test surrounding integer points of R to decide whether the subsystem contains integer points.
9. Use integer programming to decide whether the subsystem is empty.

This new dependence test has two independent subresults. First, the test includes in the system of dependence inequalities the information arising from a program's predicates. Second, it is an improved dependence test for equations with -1, 0, 1 coefficients: It is more accurate than previous tests and more efficient than integer programming. However, the subdomain dependence testing algorithm is more expensive than some previous dependence tests, e.g. the GCD test [6]. Therefore, the GCD test and other fast dependence tests can be used first to filter out some independent computations. The subdomain dependence test can then be used between the remaining dependent statements to report more accurate dependences.

3.4 Application of the Subdomain Dependence Test

3.4.1 Cycle Breaking

Dependence cycles can be removed by several techniques, e.g. *index set splitting* [7,110,113], IF removal, scalar expansion and recurrence recognition [25]. We now show that the subdomain dependence test can be used to improve index set splitting for more general cycle breaking.

Banerjee's cycle breaking technique is capable of breaking a cycle consisting of a flow and an anti-dependence between two statements in a single level loop of the following form, where B is a one-dimensional array and l, u and c_k for $1 \leq k \leq 4$ are integer constants:

Loop Nest 3.4

```
DO ($i = l, u$) {
    $S_1 : A(\ldots) = B(c_1 i + c_2)$
    $S_2 : B(c_3 i + c_4) = \ldots$      }
```

This kind of cycle can always be removed because the flow and anti-dependences are over disjoint sub-index domains of the two statements. The boundaries of the sub-index domains can be computed only when c_1 and c_3 are not both zero [7], i.e. the inverse of either $(c_1 i + c_2)$ or $(c_3 i + c_4)$ is well defined. Consider the following program from [113]:

Loop Nest 3.5

```
DO ($i = 1, 101$) {
    $S_1 : A(i) = B(101 - i) + i$
    $S_2 : B(i) = E(i)$            }
```

The flow dependence $(S_2@j \Rightarrow S_1@i)$ is over $(i \in [51, 100])$ and $(j \in [1, 50])$; and the anti-dependence $(S_1@i \Rightarrow S_2@j)$ is over $(i \in [1, 50])$ and $(j \in [51, 100])$. By splitting index domain $[1, 101]$ into disjoint domains $[1, 50]$ and $[51, 101]$, the cycle is removed and the loop can be fully parallelized as shown below:

Loop Nest 3.6

```
DOALL ($i = 1,50$) {
  $S_1 : A(i) = B(101 - i) + i$
  $S_2 : B(i) = E(i)$          }
DOALL ($i = 51,101$) {
  $S_1 : A(i) = B(101 - i) + i$
  $S_2 : B(i) = E(i)$          }
```

The subdomain dependence test breaks cycles for more general cases. It can break cycles for any number of dependences between multiple statements in a multi-level loop with reference to multi-dimensional arrays. We use the following example to show the power of the subdomain dependence test in breaking cycles:

Loop Nest 3.7

```
DO ($I$:$D$) {
  ...
  $S_1$ : IF($P_1(I)$) $A(X(I)) = C(Y(I))$
  $S_2$ : IF($P_2(I)$) $B(\ldots) = A(Z(I))$
  $S_3$ : IF($P_3(I)$) $C(W(I)) = B(\ldots)$
  ...                                   }
```

Assume there are cyclic flow dependences ($S_1 \Rightarrow S_2 \Rightarrow S_3 \Rightarrow S_1$). We know that ($S_1@I \Rightarrow S_2@J$) for I and J in the following polytope:

$$I \in (D{\downarrow}P_1), \tag{3.12}$$

$$J \in (D{\downarrow}P_2), \tag{3.13}$$

$$X(I) = Z(J), \quad \text{and} \tag{3.14}$$

$$I \preceq J. \tag{3.15}$$

Similarly, $(S_3@K \Rightarrow S_1@I)$ for I and K in the following polytope:

$$\begin{aligned} I &\in (D{\downarrow}P_1), &(3.16)\\ K &\in (D{\downarrow}P_3), &(3.17)\\ Y(I) &= W(K), \quad \text{and} &(3.18)\\ I &\succeq K. &(3.19)\end{aligned}$$

Let D_1 be the domain containing all I satisfying Equations (3.12), (3.13), (3.14) and (3.15), and let D_2 be the domain containing all I satisfying Equations (3.16), (3.17), (3.18) and (3.19). By the subdomain dependence test, we know whether integer tuples I, J and K exist which satisfy Equations from (3.12) to (3.19). If I_1, J_1 and K_1 are such integer tuples, then $(I_1 \in D_1)$ and $(I_1 \in D_2)$; therefore, D_1 and D_2 are not disjoint. Conversely, if no such integer tuples exist, then D_1 and D_2 are disjoint, and we can decompose S_1 into two sub-statements S_{1a} and S_{1b}:

$$\begin{array}{lll} S_{1a}: & \mathsf{IF}(I \in (D{\downarrow}P_1) - D_2) & A(X(I)) = \ldots \\ S_{1b}: & \mathsf{IF}(I \in (D_2)) & A(X(I)) = \ldots \end{array}$$

such that the flow dependences become $(S_{1a} \Rightarrow S_2 \Rightarrow S_3 \Rightarrow S_{1b})$, and the cycle is removed.

In order to obtain D_2, the inverse of the subscript function W should be well defined, the same as in Banerjee's technique. Let W^{-1} be the inverse of W. Equation (3.18) implies that $(K = W^{-1}(Y(I)))$ and it is clear that

$$\begin{aligned} D_2 = \{I \mid\ & I \in (D{\downarrow}P_1), (W^{-1}(Y(I)) \in (D{\downarrow}P_3)),\\ & (I \succeq W^{-1}(Y(I)))\}.\end{aligned}$$

With D_2, statement S_1 can be decomposed and the cycle can be removed.

3.4.2 Use of the Subdomain Dependence Test in Functional Programs

The subdomain dependence test is useful not only for imperative programs but also for functional programs. We use the following notation to express a functional program, where A and B are two function definitions, D and E are index domains, P_1 and P_2 are conditional expressions and Y is an affine function:

$$\begin{aligned} A(I{:}D) &= \begin{Bmatrix} P_1 \rightarrow \ldots \\ \ldots \end{Bmatrix} \\ B(I{:}E) &= \begin{Bmatrix} P_2 \rightarrow \tau[A(Y(I))] \\ \ldots \end{Bmatrix} \\ \ldots \end{aligned}$$

If only dependences between function definitions, e.g. A and B, are considered, then no dependence test is required to say that B depends on A. However, A and B can be decomposed into sub-definitions according to conditional expressions as

$$\begin{aligned} A_1(I{:}(D{\downarrow}P_1)) &= \ldots \\ B_1(I{:}(E{\downarrow}P_2)) &= \tau[A(Y(I))] \\ &\ldots \end{aligned}$$

More precise dependences can be obtained between these sub-definitions. To know if B_1 depends on A_1, the subdomain dependence test can be used to determine if the following polytope contains integer points:

$$(D{\downarrow}P_1) \cap Y(E{\downarrow}P_2).$$

Due to functionality, only one polytope, instead of $(2d+1)$, needs to be tested to obtain dependence information between each pair of sub-definitions.

Chapter 4

Static Scheduler

4.1 Motivation

For loops with true dependences which prevent large-scale parallelization, loop transformation can be applied to reveal parallelism. This chapter presents new loop transformation techniques that can extract more parallelism than previous techniques.

Organization of the Chapter In Section 4.2, we present a formal mathematical framework which unifies the previous loop transformation techniques, and sets the stage for discussing the more general classes of *loop transformers.* A *loop transformer* is a function that relates a given loop nest with its transformed version, and consists of two parts: a *spatial morphism*, and a *temporal morphism*, called a *schedule.* Next, in Section 4.3, we classify schedules by the properties of *uniformity* and the degree of parallelism to be gained, and describe the functional forms of the schedules for each class. Previous loop transformation techniques are given as examples of these classes of schedules.

In Section 4.4, we review Quinton's algorithm for obtaining *single-sequential level uniform* schedules and present the problem formulations for two new classes of schedules, namely, *subdomain-variant schedules* and *statement-variant schedules* and the algorithms to generate them. The generation of subdomain-variant schedules requires non-linear programming, and an alternative heuristic algorithm using linear programming is given.

Section 4.5 describes an iterative algorithm to obtain *multiple-sequential level* schedules based on the algorithms for single-sequential level schedules. Section 4.6 presents a recursive algorithm to generate *mixed* schedules that result in imperfectly nested loops, again using the algorithms for single-sequential level schedules as the basic step.

Finally, we illustrate the usefulness of the new loop transformation techniques with example programs in Section 4.8. Versions of the transformed program using different schedules are implemented on a Connection Machine CM/2. The difference in performance, which is essentially due to the available parallelism determined by the schedule, can amount to two orders of magnitude.

Previous Work Numerous techniques such as statement reordering, loop vectorization, interchange, permutation and skewing used in restructuring compilers [3,4,5,6,7,8,9,10,12,24,58,61,72,74,86,108,109,110,113] have been proven effective in gaining parallelism for vector computers and small-scale shared memory parallel machines.

Much work in the area of mapping recurrence equations to systolic architectures [19,29,51,59,66,81,83,84,85,88,89,90,91], in contrast, focuses on developing algorithms for loop skewing.

Guerra [41], Lin *et al.* [75] and Sheu *et al.* [103] discussed different transformation functions over different subdomains of the the iteration space of a loop nest, which is similar to the subdomain-variant schedules presented here. However, Guerra's and Lin's methods are not systematic, and Sheu's algorithm is only suitable for loops with constant dependence vectors and with partitions on the innermost loop. On the other hand, our techniques are systematic and applicable to much more general loops where the only requirement is that conditionals as well as the array index expressions are affine expressions of the loop indices.

Mauras *et al.* [79] discussed variable-variant schedules where each variable of the system of affine recurrence equations can be scheduled differently from other variables. Our statement-variant schedule, developed at about the same time [76,77], is more general than their variable-variant schedule in that our techniques are applicable to FORTRAN loops with side effects while theirs are only applicable to affine recurrence equations without side effects.

In all previous work on loop transformation, dependence vectors and dependence direction vectors are all that are needed. And for the type of loop nests of interest, there are constant numbers of such vectors. In order to generate subdomain and statement-variant schedules, we need actually to capture the *dependence index pair* where a dependence relation occurs. The problem is that there are many such pairs that need to be considered, and they can be infinitely many when the loop bounds are unknown at compile time. We need to rely on a technique called *polyhedra decomposition* [35,90,100] to manage the complexity of the algorithm.

Some recent work attempting to formalize loop transformations requires the transformation functions to be *unimodular* [10]. We will show that this requirement is not essential, and allow a much more general class of functions to be used as a loop transformer.

4.1.1 Application Domain: Affine Loops

The new loop transformation techniques presented in this chapter are applicable to a specific class of loops that consists of perfectly nested loops, possibly with conditional statements where the guards as well as the array index expressions are affine expressions of the loop indices. We call the loops in this class *affine loops.*

An Example We now use an example program to show why subdomain-variant schedules are more powerful than loop interchange and skewing. The following example is the same as Loop Nest 1.1 in Chapter 1 except that the vectorizable k loop has been removed.

Loop Nest 4.1

```
$n = 1000$
DO $(i = n, 1, -1)$ {
   DO $(j = i, n)$ {
      $S_1$ : IF$(i = j)$ $A(i,j) = B(i+1,n)$
      $S_2$ : IF$(i < j)$ $A(i,j) = A(i+1,j)$
      $S_3$ : IF$(2j - i > n)$ $B(i,j) = B(i,j-1) + A(i,j)$ } }
```

The dependence graph with dependence vectors of this program is shown in Figure 4.1. Figure 4.2(a) shows the index domain of the loop nest with the loop bound n being replaced by 5 for easier representation. An edge from domain element (i_1, j_1) to (i_2, j_2) in Figure 4.2(a) indicates that there exists statements S_a and S_b, $a \in [1,3]$, $b \in [1,3]$, such that dependence $S_a@(i_1,j_1) \Rightarrow S_b@(i_2,j_2)$ holds. Figure 4.2(b) shows dependence vectors originated from the origin for this program. Due to the strange angles of these dependence vectors, we can only skew index j with respect to index i by a factor of -1000 and then interchange loops i and j to make the inner i loop parallelizable. (The *separating hyperplane* [51] is $(-1000i + j = 0)$ so that all dependence vectors are on the same side of this hyperplane.) The combination of the skewing and interchange is a unimodular transformation which maps (i, j) to $(-1000i + j, i)$. The transformed loop nest using Wolfe's transformation algorithm [110,113] is Loop Nest 4.2 below.

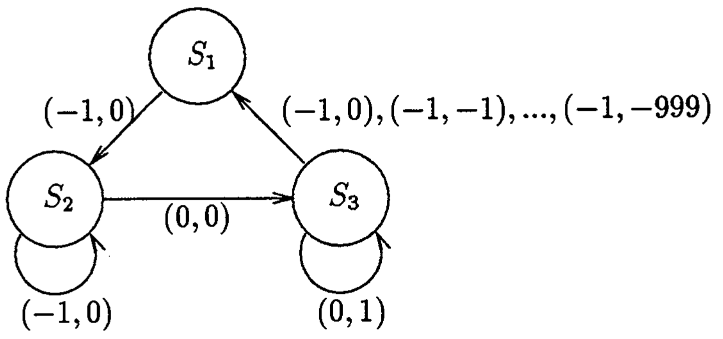

Figure 4.1: Dependence graph with dependence vectors.

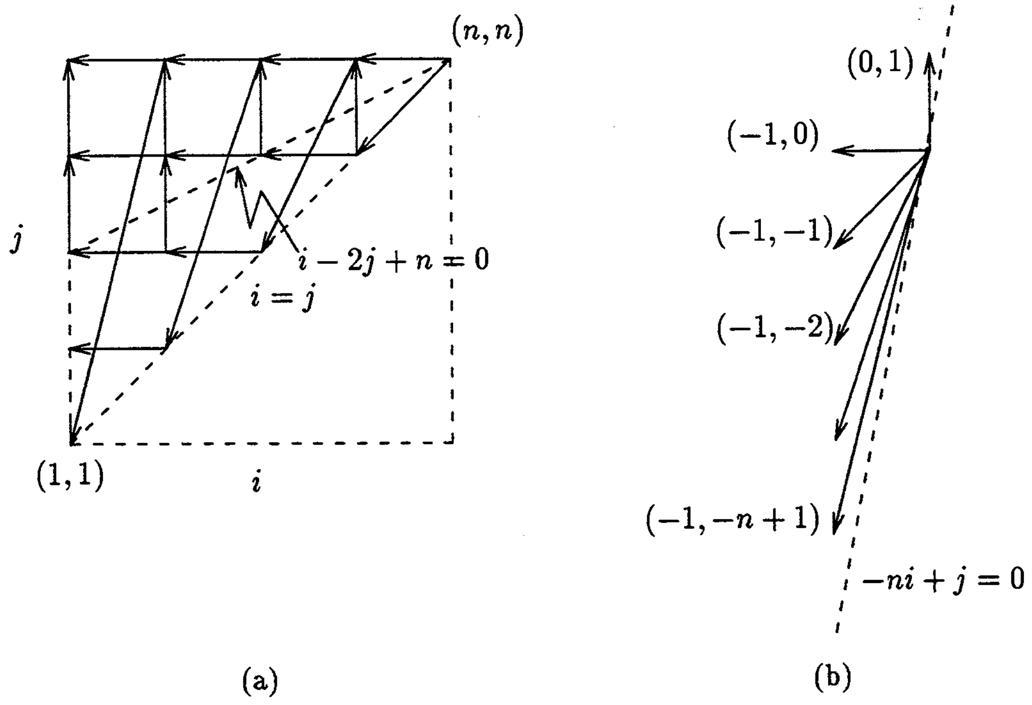

Figure 4.2: Index domain and dependence relation.

Loop Nest 4.2

```
$n = 1000$
DO $(t = -999000, 0)$ {
    DOALL $(i = (-t/1000) + 1, \max(1, -t/999), -1)$ {
        $j = t + 1000i$
        $S_1 :$ IF$(i = j)\, A(i,j) = B(i+1, n)$
        $S_2 :$ IF$(i < j)\, A(i,j) = A(i+1, j)$
        $S_3 :$ IF$(2j - i > n)\, B(i,j) = B(i, j-1) + A(i,j)$ } }
```

A better way to parallelize this program is to use a subdomain schedule where the starting index points of dependence vectors $(-1,-1)$, $(-1,-2)$, ..., $(-1,-999)$ are scheduled differently from the ending index points of these dependence vectors. This can avoid the strange angles of these dependence vectors in order to increase parallelism in the transformed loop nest.

By using the subdomain scheduling algorithm described in Section 4.4.3 below, we can obtain the following subdomain schedule with two transformations over two disjoint subdomains partitioned by the hyperplane $(2j - i = 1000)$:

$$\left\{\begin{array}{l} 2j - i > 1000 \rightarrow (-2i + j, i) = (t, i) \\ 2j - i \leq 1000 \rightarrow (-i - j + 1000, i) = (t, i) \end{array}\right\}$$

The transformed loop nest from the subdomain transformation is Loop Nest 4.3 below. Note that Loop Nest 4.3 has 1999 (= 998 + 1000 + 1) sequential steps while Loop Nest 4.2 has 999001 (= 999000 + 1) sequential steps.

Loop Nest 4.3

```
$n = 1000$
DO $(t = -1000, 998)$ {
  DOALL $(i = (1000 - t)/2, \max(1, -t), -1)$ {
    $j = t + 2i$
    IF$(2j - i > 1000)$
        $S_1$ : IF$(i = j)\, A(i,j) = B(i+1,n)$
        $S_2$ : IF$(i < j)\, A(i,j) = A(i+1,j)$
        $S_3$ : IF$(2j - i > n)\, B(i,j) = B(i,j-1) + A(i,j)$
    $j = 1000 - t - i$
    IF$(2j - i \leq 1000)$
        $S_1$ : IF$(i = j)\, A(i,j) = B(i+1,n)$
        $S_2$ : IF$(i < j)\, A(i,j) = A(i+1,j)$
        $S_3$ : IF$(2j - i > n)\, B(i,j) = B(i,j-1) + A(i,j)$ } }
```

Since there are two transformations, Wolfe's transformation algorithm for loop interchange and skewing [110,113] should be modified as follows to derive the transformed loop nests for subdomain schedules.

Loop body Each statement in the source code becomes two statements, one for each transformation, in Loop Nest 4.3. That is, the transformed loop nest has two sub-loop bodies corresponding to two transformations. Each sub-loop body should be guarded by the conditionals specifying its subdomain to restrict the computation. That is, conditional $(2j - i > 1000)$ is necessary for the sub-loop body over subdomain $(2j - i > 1000)$, and conditional $(2j - i \leq 1000)$ is required for the sub-loop body over subdomain $(2j - i \leq 1000)$. Two additional statements $(j = t + 2i)$ and $(j = 1000 - t - i)$ are needed to compute the inverses of the transformations for obtaining the original loop index.

Loop bounds If the loop bounds from these two transformations are different, then the union of these loop bounds becomes the loop bound of the transformed loop nest. For example, if the two loop bounds are $(t = a, b)$ and $(t = c, d)$, then the resulting loop bound is $t = \mathsf{min}(a, c), \mathsf{max}(b, d)$. Clearly, the sub-loop body under the transformation with $(t = a, b)$ must be guarded by an extra conditional $(a \leq t \leq b)$, and the sub-loop body under the transformation with $(t = c, d)$ must be guarded by $(c \leq t \leq d)$ to restrict the computation. On the other hand, if the loop bounds from different transformations are the same, then the resulting loop bound is also the same and no extra conditional is necessary. This special case of subdomain transformation is called *folding* [18,20] where one half of the domain is folded into the other and then the contracted domain is transformed uniformly. This happens to be the case of this example.

Loop Nest 4.3 can be simplified by removing useless statements guarded by contradictory conditionals. For example, statement S_1 in subdomain $(2j - i > 1000)$ is useless because the set of conditionals $\{(2j - i > 1000), (i = j), (i \leq j \leq 1000)\}$ is contradictory. Similarly, statement S_3 in subdomain $(2j - i \leq 1000)$ is useless. Contradictory conditional can be detected by the subdomain dependence test described in Chapter 3. After removing these two statements, Loop Nest 4.3 becomes:

Loop Nest 4.4

```
$n = 1000$
DO $(t = -1000, 998)$ {
    DOALL $(i = (1000 - t)/2, \max(1, -t), -1)$ {
        $j = t + 2i$
        IF$(2j - i > 1000)$
            $S_2$ : IF$(i < j)$ $A(i,j) = A(i+1,j)$
            $S_3$ : IF$(2j - i > n)$ $B(i,j) = B(i,j-1) + A(i,j)$
        $j = 1000 - t - i$
        IF$(2j - i \leq 1000)$
            $S_1$ : IF$(i = j)$ $A(i,j) = B(i+1,n)$
            $S_2$ : IF$(i < j)$ $A(i,j) = A(i+1,j)$            } }
```

4.2 Formalizing Loop Transformation

We now formalize the notion of loop transformation from a source loop nest to a target parallel loop nest. A *loop transformer* is a function defined over the Cartesian product of the iteration space of the loop nest and the set of statements in the body of the loop that relates a given loop nest with its transformed version. From the standpoint of symbolic transformation of the program text, a loop transformer can be decomposed into two components: the first component, called *domain morphism*, defines how the iteration space should be mapped to a new one (with new loop bounds and possibly new predicates guarding the loop body), and the second component, called *statement reordering function*, defines the ordering of the statements in the transformed loop nest. The process of obtaining a loop transformer, however, suggests another decomposition: a *temporal morphism* and a *spatial morphism*.

4.2.1 Loop Transformer and Schedule

Kinds of Index Domains For the purpose of loop transformation, it is useful to indicate how the index domain shall be interpreted. We do this by defining *kinds* of index domains. The kind of an interval domain D can be either *spatial* or *temporal*. The kind of a product domain is the product of the kinds of the component domains. For example, $D_1 \times D_2$ is of kind temporal×spatial if D_1 is of kind temporal and D_2 is of kind spatial. A single-level loop with a temporal index domain corresponds to a sequential loop (i.e. DO), while a spatial index domain corresponds to a parallel loop (i.e. DOALL).

Domain Morphism We define a *domain morphism* to be a bijective function g from index domain D to index domain E, denoted by $g\colon D \rightarrow E$, such that for all dependences $S_1@I \Rightarrow S_2@J$, condition $g(J) - g(I) \succeq \hat{0}$ holds. In other words, a domain morphism will never reverse the ordering imposed by dependence relations.

In this chapter, we assume that the target parallel machines are fine-grain. Therefore, all parallel loops are innermost loops in the transformed loop nest. That is, the codomain E of a domain morphism is a cross product of a temporal index domain E_t and a spatial index domain E_s, i.e. $E = E_t \times E_s$. Since our techniques transform one level of the source loop nest at a time, they can be easily incorporated into the algorithms for parallelizing outer loops targeted to distributed memory and large-scale multiprocessing machines.

We define g_t and g_s to be two functions:

$$g_t : D \rightarrow E_t, \quad \text{(called a \textit{temporal morphism}) and} \tag{4.1}$$

$$g_s : D \rightarrow E_s. \quad \text{(called a \textit{spatial morphism})} \tag{4.2}$$

Under domain morphism g, index I in the original loop will be mapped to index $J = g(I)$ in the transformed loop nest. Since g is bijective, it has a well-defined inverse, denoted by g^{-1}. Clearly, $I = g^{-1}(J)$. The following loop nest

Loop Nest 4.5

```
DO $((I{:}D))$ {
    ... $A(X(I))$ ... }
```

will be transformed into the following new loop nest under domain morphism $g{:}D \rightarrow E_t \times E_s$:

Loop Nest 4.6

```
DO $((J_1{:}E_t))$ {
    DOALL $((J_2{:}E_s))$ {
        ... $A(X(g^{-1}\begin{bmatrix} J_1 \\ J_2 \end{bmatrix}))$ ... } }
```

where $\begin{bmatrix} J_1 \\ J_2 \end{bmatrix}$ denotes the vertical concatenation of two column vectors J_1 and J_2 as defined in Chapter 2.

The requirement of g to be surjective is in fact not essential. For any injective function $g'{:}D \rightarrow E$, we can always derive a corresponding bijective function $g{:}D \rightarrow \{g'(I) \mid I \in D\}$ from D to the image of D under g' [19]. Therefore, by allowing the codomain of a bijective function to be the image of an injective function, we allow a much more general class of functions to be used as domain morphism. For comparison, the unimodular transformations discussed in [10,108] are special classes of bijective functions. The generality does require some nontrivial algebraic manipulation to generate correct loop bounds and predicates to guard the conditional statements in the transformed loop nest. An automatic transformation procedure for doing this based on an equational theory is described in [19].

Statement Reordering We now discuss statement reordering. Let $\mathcal{S}$ denote the set of statements in the loop body. We define a *statement reordering* to be a function r from the set of statements to the set of statement labels:

$$r{:}\mathcal{S} \rightarrow [1, s], \text{where } s = |\mathcal{S}|, \text{the number of statements in } \mathcal{S}. \qquad (4.3)$$

Loop Transformer With g and r defined above, the following function h, called the *loop transformer*, specifies how a loop nest is transformed:

$$\begin{aligned} &h{:}D \times \mathcal{S} \rightarrow E_t \times E_s \times [1,s] \\ &h(I,S) = (g_t(I), g_s(I), r(S)). \end{aligned} \tag{4.4}$$

Schedule Given h defined above, a *schedule* π is defined to be a function

$$\begin{aligned} &\pi{:}D \times \mathcal{S} \rightarrow E_t \times [1,s] \\ &\pi(I,S) = (g_t(I), r(S)), \end{aligned} \tag{4.5}$$

such that condition $\pi(J,S_2) - \pi(I,S_1) \succ \hat{0}$ must hold for all dependences $S_1@I \Rightarrow S_2@J$ in the loop nest. The condition ensures that the ordering imposed by dependence relations is preserved. Clearly, a schedule determines the sequential execution of the transformed parallel loop nest. Note that by the definition of domain morphism, $g_t(J) - g_t(I)$ can be equal to the zero vector, i.e. $S_1@I$ and $S_2@J$ can be computed at the same iteration in the transformed loop nest. In this case, statement S_1 must be in front of statement S_2 in the loop body, i.e. condition $r(S_1) < r(S_2)$ must hold, to preserve the dependence ordering.

4.2.2 Overall Procedure to Obtain a New Loop Nest

Finding a Schedule Finding a schedule π is to understand what is the potential parallelism that can be extracted from the source program. There may be alternative schedules which are incomparable without on a target machine model. Traditional loop transformation uses loop interchange to find all possible loop orderings and chooses the best one from them for the target machine [110]. Recently, Wolf and Lam [108,109] have proposed the notion of *fully permutable loop nests* as a canonical form to exploit coarse and/or fine-grain parallelism for different target machines. In their papers, unimodular loop transformations are used to find the fully permutable loop nests.

In this chapter, we focus on new techniques beyond unimodular loop transformations for parallelizing more inner loops for fine-grain parallel machines. Since our techniques schedule one level of the source loop nest at a time, they can be easily incorporated into the algorithms for parallelizing outer loops and other combinations of parallel and sequential loops, and into the algorithm for finding fully permutable loop nests.

Finding a Spatial Morphism The so-called *strip mining* and *tiling* [110,112,108] of loops are captured by the spatial morphism g_s. The choice of g_s, which depends on factors such as memory and processor organization, can be dealt with separately and is not included in this thesis. However, given a schedule $\pi = (g_t, r)$, a valid g_s should keep a loop transformer $h = (g_t, g_s, r)$ injective. In this chapter, we use the following default spatial morphism for all the examples: $g_s(i_1, \ldots, i_d) = (i_{p_1}, \ldots, i_{p_n})$, so as to result in a loop transformer h that is injective, where n is the dimensionality of the spatial index domain E_s, $\{p_1, \ldots, p_n\}$ is a subset of interval domain $[1, d]$, and $p_1 < \ldots < p_n$.

Overall Procedure To summarize, the overall procedure to obtain a new loop nest is:

1. First generate a schedule $\pi = (g_t, r)$ to maximize the degree of parallelism in inner loops, i.e. the number of parallel inner loops.

2. Then determine the spatial morphism g_s of domain morphism based on target machine characteristics such as memory and processor organization, communication cost, etc., or use a default function as shown above.

3. The loop transformer is simply $h = (g_t, g_s, r)$.

4. Finally perform symbolic program transformation, given the source loop nest and loop transformer h, to obtain the new loop nest.

The remainder of this chapter is devoted to generating π to gain large scale parallelism.

4.3 Classes of Affine and Piecewise Affine Schedules

We call a schedule affine if it is an affine function of the loop indices. We call a schedule piecewise affine if the restriction of the function to each subdomain of D and each subset of $\mathcal{S}$ is affine. In the loop restructuring literature, only affine schedules are considered. In this chapter, we consider, in addition, piecewise affine schedules.

In order to discuss the algorithms for generating suitable schedules, we now classify them according to two properties: (1) the uniformity of the schedule with respect to the set of statements $\mathcal{S}$ and the index domain D, and (2) the degree of parallelism in the transformed Loop Nest. These two properties capture the regularity of schedules. Practically speaking, the more regular a schedule the easier it is to realize.

4.3.1 Properties of Schedules

Uniformity Let index domain D be partitioned into m disjoint subdomains D_k, $1 \leq k \leq m$; and let the set of statements $\mathcal{S}$ be partitioned into n disjoint subsets $\mathcal{S}_k$, $1 \leq k \leq n$. The general form of a piecewise affine schedule π defined in Equation (4.5) consists of conditional branches, one for each pair of subdomain D_i and statement subset $\mathcal{S}_j$, and an affine expression of the loop indices is on the right-hand side of each branch. We call a schedule

1. *uniform* if $m = 1$ and $n = 1$,
2. *subdomain-variant* if $m > 1$ and $n = 1$, (also called a subdomain schedule)
3. *statement-variant* if $m = 1$ and $n > 1$, or

4. *nonuniform* if $m > 1$ and $n > 1$.

Degree of Generated Parallelism As defined in Equations (4.2) and (4.5), the dimensionality of E_t, the temporal index domain, indicates the number of levels of sequential loops in the transformed loop nest. Hence a schedule π would generate a target loop nest with more levels of parallel loops and thus potentially more parallelism if E_t is of lower dimensionality. We call the dimensionality of E_t the *sequential level* of π. Schedules can thus be classified as:

1. *Single-sequential level schedule* if E_t is a subset of the set of natural numbers $\mathcal{N}$.
2. *Multiple-sequential level schedule* if E_t is a subset of $\mathcal{N}^n$, where n is a positive integer and $n \leq d$, the dimensionality of the original loop nest.
3. *Mixed schedule* if E_t can be of different dimensions for each pair of subdomain D_i and statement subset $\mathcal{S}_j$. Such a mixed schedule will result in transformed programs consisting of imperfectly nested loops.

Loop vectorization, interchange, permutation, reversal, skewing and unimodular transformations are examples of single-sequential level and multiple-sequential level uniform schedules. And inner-loop fission is an example of mixed schedules.

4.3.2 Classification and Functional Form of Schedules

Classification Clearly, the uniformity of π and the dimensionality of π are two orthogonal properties, except that a mixed schedule cannot be uniform. Thus there are all together eleven $(4 * 3 - 1)$ classes of affine and piecewise affine schedules. The classes and their acronyms ranging from single-sequential level uniform schedules to mixed nonuniform schedules are given in Table 4.1.

	Single-Sequential Level (SSL)	Multiple-Sequential Level (MSL)	Mixed
Uniform (U)	SSL-U	MSL-U	
Subdomain (SD)	SSL-SD	MSL-SD	Mixed-SD
Statement-Variant (SV)	SSL-SV	MSL-SV	Mixed-SV
Nonuniform (NU)	SSL-NU	MSL-NU	Mixed-NU

Table 4.1: Classes of static schedulers.

Functional Form We now describe the forms of affine and piecewise affine schedules by using matrix and vector notations. Let $r(S)$ for a given S in $\mathcal{S}$ be a constant scalar. Let d be the dimensionality of the index domain of the source loop nest.

Uniform Schedule:

$$\pi(I,S) = (TI, r(S)), \qquad I \in D, S \in \mathcal{S}, \tag{4.6}$$

where T is a constant l-by-d matrix and l is the sequential level of the schedule π.

Subdomain Schedule:

$$\pi(I,S) = \left\{\begin{array}{l} I \in D_1 \rightarrow (T_1 \begin{bmatrix} I \\ 1 \end{bmatrix}, r_1(S)) \\ \dots \\ I \in D_m \rightarrow (T_m \begin{bmatrix} I \\ 1 \end{bmatrix}, r_m(S)) \end{array}\right\}, \qquad I \in D, S \in \mathcal{S}, \tag{4.7}$$

where T_i, $1 \leq i \leq m$, is a constant l_i-by-$(d+1)$ matrix, l_i is the sequential level of the part of the schedule defined over D_i, and $\begin{bmatrix} I \\ 1 \end{bmatrix}$ denotes the vertical concatenation of a column vector I and a degenerate vector with one element of value 1.

Statement-Variant Schedule:

$$\pi(I,S) = \left\{ \begin{array}{l} S \in \mathcal{S}_1 \rightarrow (T_1 \begin{bmatrix} I \\ 1 \end{bmatrix}, r(S)) \\ \ldots \\ S \in \mathcal{S}_n \rightarrow (T_n \begin{bmatrix} I \\ 1 \end{bmatrix}, r(S)) \end{array} \right\}, \quad I \in D, S \in \mathcal{S}, \tag{4.8}$$

where T_i, $1 \leq i \leq n$, is a constant l_i-by-$(d+1)$ matrix and l_i is the sequential level of the part of the schedule defined over $\mathcal{S}_i$.

Nonuniform Schedule:

$$\pi(I,S) = \left\{ \begin{array}{l} (I \in D_1) \rightarrow \left\{ \begin{array}{l} (S \in \mathcal{S}_1) \rightarrow (T_{11} \begin{bmatrix} I \\ 1 \end{bmatrix}, r_1(S)) \\ \ldots \\ (S \in \mathcal{S}_n) \rightarrow (T_{1n} \begin{bmatrix} I \\ 1 \end{bmatrix}, r_1(S)) \end{array} \right\} \\ \ldots \\ (I \in D_m) \rightarrow \left\{ \begin{array}{l} (S \in \mathcal{S}_1) \rightarrow (T_{m1} \begin{bmatrix} I \\ 1 \end{bmatrix}, r_m(S)) \\ \ldots \\ (S \in \mathcal{S}_n) \rightarrow (T_{mn} \begin{bmatrix} I \\ 1 \end{bmatrix}, r_m(S)) \end{array} \right\} \end{array} \right\}, \quad I \in D, S \in \mathcal{S},$$

where T_{ij}, $1 \leq i \leq m$ and $1 \leq j \leq n$, is a constant l_{ij}-by-$(d+1)$ matrix and l_{ij} is the sequential level of the part of the schedule defined over D_i and $\mathcal{S}_j$.

The linear terms TI and $T \begin{bmatrix} I \\ 1 \end{bmatrix}$, $I \in D$, determine the form of the sequential loops in the transformed loop nest, which includes nesting structures, bounds, and possibly additional predicates to guard the loop body. The constant terms $r(S)$ determine the orders of the statements in the transformed loop body.

4.3.3 Examples of Different Classes of Schedules

We now give some examples of different classes of schedules. We first show that loop vectorization, interchange, permutation and skewing are special cases of multiple-sequential uniform schedules.

In the following, we use loc as a function that returns the position of the statement S in the source loop nest:

$$\begin{aligned} \mathsf{loc} &: \mathcal{S} \rightarrow \mathcal{N} \\ \mathsf{loc}(S) &= \text{the position of the statement } S \text{ in the source loop body} \end{aligned} \tag{4.9}$$

We define $V(k)$ as a vector of length d with k-th element being 1 and all other elements being 0, where d is the dimension of the loop nest.

Example 1: Loop Vectorization Suppose a dependence test says that m innermost loops can be parallelized. The schedule for the so-called loop vectorization of the $d-m$ outermost loops is of the following form, where d is the dimensionality of the index domain of the loop nest:

$$\pi(I,S) = (i_1, i_2, \ldots, i_{d-m}, \mathsf{loc}(S)), \tag{4.10}$$

$$\text{i.e.} \qquad T = \begin{pmatrix} V(1) \\ \ldots \\ V(d-m) \end{pmatrix}, \text{ and} \tag{4.11}$$

$$r(S) = \mathsf{loc}(S). \tag{4.12}$$

Example 2: Loop Interchange and Permutation Loop interchange and permutation [3,4,5,9,110,111,113] is a process of switching inner and outer loops. We use Loop Nest 2.1 as an example to show the effect of loop interchange and permutation. For easy reference, Loop Nest 2.1 is repeated below:

Loop Nest 4.7

```
DO ($i_1 = l_1, u_1$) {
  DO (...) {
    DO ($i_d = l_d, u_d$) {
      loop body }    } }
```

Suppose Loop Nest 4.7 after loop interchange or permutation becomes

Loop Nest 4.8

```
DO ($i_{p_1} = l_{p_1}, u_{p_1}$) {
  DO (...) {
    DO ($i_{p_d} = l_{p_d}, u_{p_d}$) {
      body }          } },
```

where $(p_1, p_2, \ldots, p_d)$ is a permutation of $(1, 2, \ldots, d)$. Also suppose the m innermost loops are parallelizable. The schedule π has the form:

$$\pi(I, S) = (i_{p_1}, i_{p_2}, \ldots, i_{p_{d-m}}, \mathsf{loc}(S)), \tag{4.13}$$

$$\text{i.e.} \qquad T = \begin{pmatrix} V(p_1) \\ \ldots \\ V(p_{d-m}) \end{pmatrix}, \text{ and} \tag{4.14}$$

$$r(S) = \mathsf{loc}(S). \tag{4.15}$$

Example 3: Loop Skewing This operation transforms Loop Nest 4.7 as follows: shifting index i_n with respect to index i_m, $1 \leq m < n \leq d$, by a factor of f, where f is a positive integer, replacing l_n with the expression $(l_n + i_m * f)$, replacing u_n with the expression $(u_n + i_m * f)$, and replacing all occurrences of i_n in the loop body with the expression $(i_n - i_m * f)$ [110,113]. The transformed loop nest is of the form:

Loop Nest 4.9

```
DO ($i_1 = l_1, u_1$) {
  ...
  DO ($i_n = l_n + i_m * f, u_n + i_m * f$) {
    ...
    DO ($i_d = l_d, u_d$) {
      same loop body but with $i_n$ being replaced by $(i_n - i_m * f)$ } } }
```

The schedule for such so called loop skewing is of the form:

$$\pi(I, S) = (i_1, \ldots, i_m, \ldots, \underbrace{i_n + f * i_m}_{n\text{-th element}}, \ldots, i_d, \mathsf{loc}(S)), \tag{4.16}$$

$$\text{i.e.} \qquad T = \begin{pmatrix} V(1) \\ \ldots \\ V(n) + f * V(m) \\ \ldots \\ V(d) \end{pmatrix}, \text{ and} \tag{4.17}$$

$$r(S) = \mathsf{loc}(S). \tag{4.18}$$

Example 4: Single-Sequential Level Uniform Schedule

Loop Nest 4.10

```
DO ($i = 1, n$) {
  DO ($j = 1, n$) {
    $S_1 : A(i, j) = B(i, j - 1) + i$
    $S_2 : B(i, j) = A(i - 1, j) + j$ } }
```

A single-sequential level uniform schedule

$$\pi((i,j),S_1) = (i,2), \quad \text{and} \tag{4.19}$$
$$\pi((i,j),S_2) = (i,1), \tag{4.20}$$

with spatial morphism $g_s(i,j) = (j)$ will transform Loop Nest 4.10 into

Loop Nest 4.11

```
DO ($i = 1, n$) {
    DOALL ($j = 1, n$) {
        $S_2 :\ B(i,j) = A(i-1,j) + j$
        $S_1 :\ A(i,j) = B(i,j-1) + i$ } }
```

Example 5: Multiple-Sequential Level Uniform Schedule

Loop Nest 4.12

```
DO ($i = n-1, 1, -1$) {
    DO ($j = i+1, n$) {
        DO ($k = i, j$) {
            $S_1 :\ \mathsf{IF}(i+1 = k)\, B(i,j,k) = C(i+1,j,j)$
            $S_2 :\ \mathsf{IF}(i+1 < k)\, B(i,j,k) = B(i+1,j,k)$
            $S_3 :\ \mathsf{IF}(i+j+1 < 2k)\, C(i,j,k) = C(i,j,k-1) + B(i,j,k)$ } } }
```

A two-sequential level uniform schedule

$$\pi((i,j,k),S) = ((-i,k), \mathsf{loc}(S)) \tag{4.21}$$

with spatial morphism $g_s(i,j,k) = (j)$ will transform Loop Nest 4.12 into

Loop Nest 4.13

```
DO ($i = n-1, 1, -1$) {
    DO ($k = i, n$) {
        DOALL ($j = \max(i+1, k), n$) {
            $S_1$ : IF($((i+1=k))$) $B(i,j,k) = C(i+1,j,j)$
            $S_2$ : IF($((i+1<k))$) $B(i,j,k) = B(i+1,j,k)$
            $S_3$ : IF($((i+j+1<2k))$) $C(i,j,k) = C(i,j,k-1) + B(i,j,k)$ } } }
```

Example 6: Mixed Statement-Variant Schedule Consider Loop Nest 4.12 again. The following schedule

$$\pi((i,j,k),S) = \left\{ \begin{array}{l} S = S_3 \rightarrow ((i,k), \mathrm{loc}(S)) \\ \mathrm{else} \rightarrow (i, \mathrm{loc}(S)) \end{array} \right\} \tag{4.22}$$

with spatial morphism $g_s(i,j,k) = (j)$ for statement S_3 and $g_s(i,j,k) = (j,k)$ for statements S_1 and S_2 transforms Loop Nest 4.12 to Loop Nest 4.14 below, which consists of imperfectly nested loops:

Loop Nest 4.14

```
DO ($i = n-1, 1, -1$) {
    DOALL ($((j = i+1, n), (k = i, j))$) {
        $S_1$ : IF($((i+1=k))$) $B(i,j,k) = C(i+1,j,j)$
        $S_2$ : IF($((i+1<k))$) $B(i,j,k) = B(i+1,j,k)$ }
    DO ($k = i, n$) {
        DOALL ($j = \max(i+1, k), n$) {
            $S_3$ : IF($((i+j+1<2k))$)
                $C(i,j,k) = C(i,j,k-1) + B(i,j,k)$ } } }
```

Example 7: Single-Sequential Level Subdomain-Variant Schedule Another possible transformation of Loop Nest 4.12 is the schedule:

$$\pi((i,j,k),S) = \begin{Bmatrix} i+j+1 \leq 2k \rightarrow (-2i+j+k, r_1(S)) \\ i+j+1 > 2k \rightarrow (-i+2j-k, r_2(S)) \end{Bmatrix}, \tag{4.23}$$

where $r_1(S)$ and $r_2(S)$ are 1 for all $S \in \{S_1, S_2, S_3\}$ except $r_1(S_3) = 2$, which implies that statement S_3 under subdomain $(i+j+1 \leq 2k)$ should be the last statement in the loop body. Using the symbolic loop transformation methods described in Section 4.1, this schedule with spatial morphism $g_s(i,j,k) = (i,j)$ will transform Loop Nest 4.12 into:

Loop Nest 4.15

```
DO $(t = 1, 2n-2)$ {
  DOALL $(i = \min(n-1, (2n-t)/2), 1, -1)$ {
    DOALL $(j = \max(i+1, (t+2i)/2), \min(n, t+i))$ {
      $k = -t - i + 2j$
      IF$(i+j+1 > 2k)$
          $S_1$ : IF$(i+1 = k)\, B(i,j,k) = C(i+1,j,j)$
          $S_2$ : IF$(i+1 < k)\, B(i,j,k) = B(i+1,j,k)$
      $k = t + 2i - j$
      IF$(i+j+1 \leq 2k)$
          $S_1$ : IF$(i+1 = k)\, B(i,j,k) = C(i+1,j,j)$
          $S_2$ : IF$(i+1 < k)\, B(i,j,k) = B(i+1,j,k)$
          $S_3$ : IF$(i+j+1 < 2k)\, C(i,j,k) = C(i,j,k-1) + B(i,j,k)$ } } }
```

Similar to the transformed Loop Nest 4.3, each statement in Loop Nest 4.12 results in two statements in the transformed loop nest, except that statement S_3 is guarded by contradictory conditionals under subdomain $(i + j + 1 > 2k)$. In fact, Loop Nest 4.12 is part of the dynamic programming code presented in Section 4.8.

4.4 Algorithms for Generating Single-Sequential Level Schedules

Our algorithms for generating the various single-sequential level schedules are based on Quinton's algorithm for generating SSL-U schedules [35,89,90]. In the following, we first review Quinton's algorithm. We then present two algorithms, one for generating SSL-SD schedules and the other for SSL-SV schedules.

4.4.1 Previous Work: Uniform Scheduling Algorithm

Quinton's approach addresses the analysis and mapping of linear recurrence equations [35,89,90]. We formulate Quinton's algorithm in the context of loop transformations.

Problem Formulation

Constraints Derived from Data Dependences For an SSL-U schedule $\pi(I, S) = (TI, r(S))$, where T is a row vector, the inequality

$$(TJ, r(S_2)) - (TI, r(S_1)) \succ \hat{0}, \tag{4.24}$$

must hold for all index tuples I and J in index domain D and statements S_1 and S_2 in $\mathcal{S}$ such that $S_1@I \Rightarrow S_2@J$. We first focus on the problem of obtaining T satisfying the following more stringent condition:

$$TJ - TI = T(J - I) > 0 \tag{4.25}$$

for each dependence $S_1@I \Rightarrow S_2@J$. If such T exists, then Equation (4.24) also holds for all dependences $S_1@I \Rightarrow S_2@J$ due to the lexicographical ordering "$\succ$" regardless of what r is. In this case, the ordering among statements can be arbitrary. How to obtain T satisfying less stringent conditions, and use r, in addition to T, to preserve the ordering imposed by dependences, will be discussed in Section 4.4.2.

Space of Dependence Vectors It is clear that in the case of uniform schedule, dependence vectors $J - I$ are sufficient for obtaining T. We now formulate the set of dependence vectors for conditional statements.

For the rest of the chapter, we consider only flow dependence in the generic Loop Nest L given in Chapter 2. Anti-dependence and output dependence can be treated similarly. In this case, the set of dependence vectors from statement S_1 to S_2, denoted by $\mathcal{V}(S_1, S_2)$, in Loop Nest L is:

$$\mathcal{V}(S_1, S_2) = \{J - I \mid I \in (D{\downarrow}P_1), J \in (D{\downarrow}P_2), \text{ and } X(I) = Y(J)\}, \tag{4.26}$$

where $(D{\downarrow}P_1)$ is a sub-index domain of D under the restriction of predicate P_1 as defined in Equation (3.5).

Input System As described before, each dependence relation $S_1@I \Rightarrow S_2@J$ defines an inequality $T(J - I) > 0$ that row vector T must satisfy. We call the set of all constraints on T the *input system*, denoted by $\mathcal{C}$:

$$\begin{aligned} \mathcal{C} &= \{T(J-I) > 0 \mid \text{ there exist statements } S_1 \text{ and } S_2 \text{ in } \mathcal{S}, \text{ and index} \\ &\qquad\qquad\qquad\quad \text{tuples } I \text{ and } J \text{ in } D, \text{ such that } S_1@I \Rightarrow S_2@J\} \\ &= \{T(J-I) > 0 \mid \text{ there exist statements } S_1 \text{ and } S_2 \text{ in } \mathcal{S}, \\ &\qquad\qquad\qquad\quad \text{such that } (J - I) \in \mathcal{V}(S_1, S_2)\}. \end{aligned} \tag{4.27}$$

Decomposing Polyhedra into Vertices and Extremal Rays

Polyhedra Decomposition There can be many dependence vectors that need to be considered, and they can be infinitely many when the loop bounds are unknown at compile time. We need to rely on a technique that decomposes a polyhedron into *vertices* and *extremal rays* [35,90,100] to manage the complexity of the algorithm.

(As defined in Section 3.2.2, a polyhedron is a intersection of a finite number of hyperplanes and half spaces, and a polytope is a bounded polyhedron.) The algorithm for polyhedra decomposition is described in [35].

Constraints on the Input System In the loop restructuring literature, only *rectangular* and *trapezoid* index domains of loop nests are considered [8,110]. Note that rectangular and trapezoid index domains are special classes of polyhedra [78]. In order to obtain uniform schedules systematically, Quinton restricts the index domain of each recurrence equation to be a polyhedron and all subscript functions (called *index mappings* in [90]) to be affine expressions of the indices used in defining the index domains. In FORTRAN-like programs, the above restriction is translated to perfectly nested loops where loop bounds at one level may depend on the outer levels, with the loop body consisting of conditional statements where all the predicates of conditionals and all subscript functions are affine expressions of the loop indices. Under these restrictions, Quinton [90] shows that the set $\mathcal{V}(S_1, S_2)$ of the dependence vectors from statement S_1 to S_2 is a d-dimensional polyhedron, where d is the dimensionality of the index domain of the loop nest.

We now discuss how to represent polyhedron $\mathcal{V}(S_1, S_2)$ by its vertices and extremal rays.

Vertices and Extremal Rays Any polyhedron can be decomposed into a finite set of *vertices* and *extremal rays* [100]. (Since a *line* can be interpreted as two rays in opposite directions [100], vertices and extremal rays are sufficient for polyhedra decomposition.) Here, we use the following example to show what vertices and extremal rays are. For formal definitions, please refer to [100]. Consider the polyhedron specified by two inequalities: $\{(x, y) \mid 2x - y \geq 2, \text{ and } 2y - x \geq -1\}$. Point $(1, 0)$ is the vertex and two vectors $(1, 2)$ and $(2, 1)$ are the extremal rays of this polyhedron as shown in Figure 4.3.

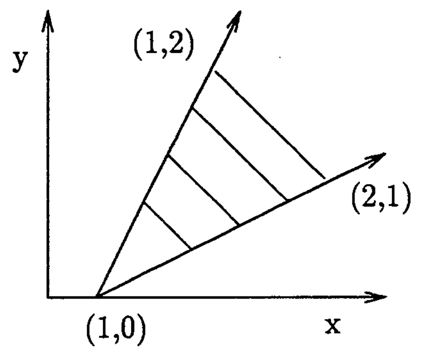

Figure 4.3: Point (1,0) is the vertex and two vectors (1,2) and (2,1) are the extremal rays.

Linear Programming Formulation

After polyhedra decomposition, each point in polyhedron $\mathcal{V}(S_1, S_2)$ can be expressed as the sum of a convex combination of the vertices and of a positive combination of the extremal rays of $\mathcal{V}(S_1, S_2)$ [100]. Based on this property, Quinton [89,90] shows that $T(J - I) > 0$ holds for all dependence vectors $J - I$ in $\mathcal{V}(S_1, S_2)$ if and only if the following two conditions hold:

$$\begin{array}{l} \text{for all vertex } V \text{ of } \mathcal{V}(S_1, S_2),\ TV > 0, \text{ and} \\ \text{for all extremal ray } R \text{ of } \mathcal{V}(S_1, S_2),\ TR \geq 0. \end{array} \tag{4.28}$$

By Quinton's theorem, the input system $\mathcal{C}$ defined in Equation (4.27) can be simplified to:

$$\begin{aligned} \mathcal{C} = \{TV > 0 \mid & \text{ for all vertex } V \text{ of } \mathcal{V}(S_1, S_2), \text{ where } S_1 \text{ and } S_2 \\ & \text{ are two statements in } \mathcal{S} \text{ such that } S_1 \Rightarrow S_2\} \\ \cup\ \{TR \geq 0 \mid & \text{ for all extremal ray } R \text{ of } \mathcal{V}(S_1, S_2), \text{ where } S_1 \text{ and } S_2 \\ & \text{ are two statements in } \mathcal{S} \text{ such that } S_1 \Rightarrow S_2\}. \end{aligned} \tag{4.29}$$

Consequently, the row vector T of length d, which defines an SSL-U schedule of a loop nest, can be obtained by linear programming, where d is the dimensionality of the index domain of the loop nest. The dimensionality of the linear programming system is d, and the number of constraints is the sum of the number of vertices and extremal rays.

4.4.2 An Algorithm for Statement Reordering

Having reviewed the algorithm for generating an SSL-U schedule with arbitrary statement reordering function r, we now discuss how to obtain an SSL-U schedule with statement reordering terms: $\pi(I, S) = (g_t(I), r(S))$, where g_t is the temporal morphism defined in Equation (4.2) and $g_t I = TI$, and r is the statement reordering function defined in Equation (4.3). The method can be modified easily to obtain the statement reordering functions for other single-sequential level schedules, i.e. SSL-SD, SSL-SV and SSL-NU.

The original input system for obtaining a schedule $\pi(I, S) = (g_t(I), r(S))$ should be:

$$\begin{aligned} \mathcal{C} = \{(g_t(J), r(S_2)) \succ (g_t(I), r(S_1)) \mid & \text{ there exist statements } S_1 \text{ and } S_2 \text{ in } \mathcal{S}, \\ & \text{and index tuples } I \text{ and } J \text{ in } D, \text{ such that } S_1@I \Rightarrow S_2@J\}. \end{aligned} \tag{4.30}$$

To solve for g_t and r separately in two steps, the formulation is developed as follows. We start with the notion of minimal target difference vectors.

Minimal Target Difference Vector We define Γ to be the function space $[D \rightarrow E_t]$, where E_t is a one-dimensional temporal index domain, and define $\mathcal{Z}$ to be the set of rationals. We define a second order function $\mu_v(f, S_1, S_2)$ to be the minimal target difference vector (in the sense of lexicographical ordering on elements of E_t) ranging over the image of the set of dependence vectors $\mathcal{V}(S_1, S_2)$ defined in Equation (4.26) under function $f \in \Gamma$:

$$\begin{aligned} \mu_{\mathcal{V}} &: \Gamma \times \mathcal{S} \times \mathcal{S} \to \mathcal{Z} \\ \mu_{\mathcal{V}}(f, S_1, S_2) &= \min\{f(K) \mid K \in \mathcal{V}(S_1, S_2)\} \\ f(I) &= TI = g_t(I). \end{aligned} \tag{4.31}$$

Input Systems for g_t and r Due to the lexicographical ordering $\succ$, condition $(g_t(J), r(S_2)) \succ (g_t(I), r(S_1))$ in Equation (4.30) holds for all $(J - I) \in \mathcal{V}(S_1, S_2)$ if and only if either $\mu_{\mathcal{V}}(g_t, S_1, S_2) > 0$ holds or both $\mu_{\mathcal{V}}(g_t, S_1, S_2) = 0$ and $r(S_2) > r(S_1)$ hold. Consequently, as far as the statement reordering function r is concerned, the dependences $S_1 \Rightarrow S_2$ where $\mu_{\mathcal{V}}(g_t, S_1, S_2) > 0$ do not provide any constraint on r. Only for those dependences $S_1 \Rightarrow S_2$ such that $\mu_{\mathcal{V}}(g_t, S_1, S_2) = 0$, the conditions $r(S_1) < r(S_2)$ must hold. So the algorithm for generating SSL schedules can start out with a less stringent criterion $\mu_{\mathcal{V}}(g_t, S_1, S_2) \geq 0$ for all pairs of statements S_1 and S_2 in $\mathcal{S}$ such that $S_1 \Rightarrow S_2$ to find g_t, and follow by the criterion $r(S_1) < r(S_1)$ for those dependence relation $S_1 \Rightarrow S_2$ where $\mu_{\mathcal{V}}(g_t, S_1, S_2) = 0$.

Note that condition $\mu_{\mathcal{V}}(g_t, S_1, S_2) \geq 0$ holds if and only if the following conditions are true:

$$\begin{aligned} &\text{for all vertex } V \text{ of } \mathcal{V}(S_1, S_2),\ TV \geq 0, \text{ and} \\ &\text{for all extremal ray } R \text{ of } \mathcal{V}(S_1, S_2),\ TR \geq 0, \end{aligned} \tag{4.32}$$

and condition $\mu_{\mathcal{V}}(g_t, S_1, S_2) = 0$ holds if and only if conditions in Equation (4.32) hold and there exists a vertex V of $\mathcal{V}(S_1, S_2)$ such that $(TV = 0)$ is true.

From the above discussion, we know that the input system defined in Equation (4.30) can be separated into two parts, one for temporal morphism g_t and the other for statement reordering function r:

$$\text{for } g_t : \quad \begin{aligned} \mathcal{C}_{g_t} = \{\mu_{\mathcal{V}}(g_t, S_1, S_2) \geq 0 \mid &\text{ there exist statements } S_1 \text{ and } S_2 \text{ in } \mathcal{S}, \\ &\text{ such that } S_1 \Rightarrow S_2\}, \end{aligned} \tag{4.33}$$

$$\text{for } r : \quad \begin{aligned} \mathcal{C}_r = \{ r(S_1) < r(S_2) \mid & \text{ there exist statements } S_1 \text{ and } S_2 \text{ in } \mathcal{S}, \\ & \text{such that } S_1 \Rightarrow S_2 \text{ and } \mu_v(g_t, S_1, S_2) = 0 \}. \end{aligned} \tag{4.34}$$

Linear programming can be used to obtain the temporal morphism g_t from the input system specified in Equation (4.33). We now discuss how to obtain statement reordering function r from the input system specified in Equation (4.34) given a temporal morphism g_t.

Partial Ordering and Topological Sort To obtain statement reordering function r given a temporal morphism g_t, we need the notion of *partial ordering.* A partial order on a set S is a binary relation "<" that is *transitive* and *irreflexive*, i.e. there cannot be any cyclic relations $x < \ldots < x$ for all elements $x \in S$. For the purpose of statement reordering, we define S to be the set of statements $\mathcal{S}$ and we say $S_1 < S_2$, i.e. statement S_1 must be in front of statement S_2 in the transformed loop body, if $S_1 \Rightarrow S_2$ and $\mu_v(g_t, S_1, S_2) = 0$.

If $\mathcal{S}$ has a partial ordering, then *topological sort* [53] can produce a linear ordering of the statements, which defines r. If $\mathcal{S}$ does not have a partial ordering, e.g. if there are cyclic relations $S_1 < S_2$ and $S_2 < S_1$, then there cannot be any r that will satisfy both $r(S_1) < r(S_2)$ and $r(S_1) > r(S_2)$. In this case, the given temporal morphism g_t should be rejected.

4.4.3 Subdomain Scheduling Algorithm

This section presents two algorithms for generating SSL-SD schedules. In the following, we first formulate the input system for finding a subdomain schedule for a given loop nest. From the input system, we can see that the hard part of finding an SSL-SD schedule is to determine where the subdomain boundaries are. We describe a heuristic algorithm that first makes guesses at possible subdomain boundaries, and then obtains an SSL-SD schedule by linear programming with given subdomains. Subdomain boundaries are chosen from predicates in conditionals or from unbounded *inside-out* enumerative search. We then present a general algorithm which searches through a bounded space for possible hyperplanes that partition the domain, and comes up with affine schedules, one for each subdomain, by nonlinear programming. However, the complexity of this general method is unacceptably high.

Problem Formulation

Constraints Derived from Data Dependences For an SSL-SD schedule defined in Equation (4.7), the inequality

$$(T_j \begin{bmatrix} J \\ 1 \end{bmatrix}, r_j(S_2)) - (T_i \begin{bmatrix} I \\ 1 \end{bmatrix}, r_i(S_1)) \succ \hat{0} \tag{4.35}$$

must hold for all index tuples $I \in D_i$ and $J \in D_j$, $1 \leq i, j \leq m$ (m is the number of subdomains), and statements S_1 and S_2 in $\mathcal{S}$ such that $S_1@I \Rightarrow S_2@J$, where T_i and T_j are row vectors of length $(d+1)$. We focus on the problem of obtaining T_i and T_j satisfying the condition

$$T_j \begin{bmatrix} J \\ 1 \end{bmatrix} - T_i \begin{bmatrix} I \\ 1 \end{bmatrix} \geq 0, \tag{4.36}$$

which is the first step in obtaining an SSL-SD schedule. This step will be followed by a topological sort to find the statement reordering function as discussed in Section 4.4.2.

Since row vector T_i can be different from T_j, Equation (4.36) cannot be rewritten as $T(J-I) \geq 0$. Consequently, dependence vectors are not adequate for obtaining subdomain schedules. A new representation of a dependence relation is necessary.

Dependence Index pairs We now introduce a new notion of dependence relations. If dependence $S_1@I \Rightarrow S_2@J$ exists, then we call (I, J) a *dependence index pair* from statement S_1 to S_2.

Space of Dependence Index Pairs Similar to the set of dependence vectors $\mathcal{V}(S_1, S_2)$ defined in Equation (4.26), we formulate the set of dependence index pairs from statement S_1 to S_2, denoted by $\mathcal{P}(S_1, S_2)$, as:

$$\mathcal{P}(S_1, S_2) = \{(I, J) \mid I \in (D{\downarrow}P_1), J \in (D{\downarrow}P_2), \text{ and } X(I) = Y(J)\}. \tag{4.37}$$

It is easy to see that, under the restrictions discussed in Section 4.4.1, $\mathcal{P}(S_1, S_2)$ is a $(2d)$-dimensional polyhedron, where d is the dimensionality of the index domain of the loop nest.

Input System for g_t The input system for obtaining an SSL-SD temporal morphism g_t consists of the following constraints:

$$\begin{aligned} \mathcal{C}_{g_t} = \{T_j \begin{bmatrix} J \\ 1 \end{bmatrix} - T_i \begin{bmatrix} I \\ 1 \end{bmatrix} \geq 0 \mid & \text{ there exist statements } S_1 \text{ and } S_2 \text{ in } \mathcal{S}, \\ & \text{such that } (I, J) \in Q(S_1, S_2, i, j)\}, \quad \text{where} \end{aligned} \tag{4.38}$$

$$Q(S_1, S_2, i, j) = \{(I, J) \mid (I, J) \in \mathcal{P}(S_1, S_2), I \in D_i, J \in D_j\}.$$

In order to obtain subdomain schedules systematically, we restrict that the subdomains are separated by hyperplanes. Under this restriction, each subdomain D_i, $1 \leq i \leq m$, is a d-dimensional polyhedron, and each $Q(S_1, S_2, i, j)$ defined in Equation (4.38) is a $(2d)$-dimensional polyhedron.

Again, we need to represent each polyhedron $Q(S_1, S_2, i, j)$ by its vertices and extremal rays. However, polyhedron $Q(S_1, S_2, i, j)$ defined in Equation (4.38) is specified by two unknown subdomains D_i and D_j. We will describe a heuristic algorithm that makes guesses at possible subdomain boundaries. Once the subdomains are determined, polyhedron $Q(S_1, S_2, i, j)$ can be represented by its vertices and extremal rays, and a subdomain schedule can be obtained by linear programming just like the way a uniform schedule is obtained.

Statement Reordering In Section 4.4.2, we discuss how to obtain a statement reordering function r given an SSL-U temporal morphism g_t. All formulations in Section 4.4.2 can be easily modified for other classes of g_t, i.e. SSL-SD, SSL-SV and SSL-NU.

Let Γ, E_t, $\mathcal{Z}$ be as defined in Section 4.4.2. Let $\mathcal{N}$ be the set of natural numbers. We define a second-order function $\mu_{SD}(f, S_1, S_2, i, j)$ to be the minimal target difference vector ranging over the vectors $f(J) - f(I)$ where $(I, J) \in Q(S_1, S_2, i, j)$ and $f \in \Gamma$:

$$\begin{aligned} \mu_{SD} &: \Gamma \times \mathcal{S} \times \mathcal{S} \times \mathcal{N} \times \mathcal{N} \to \mathcal{Z} \\ \mu_{SD}(f, S_1, S_2, i, j) &= \min\{f(J) - f(I) \mid (I, J) \in Q(S_1, S_2, i, j)\} \qquad (4.39) \\ f(I) &= T_i \begin{bmatrix} I \\ 1 \end{bmatrix} = g_t(I) \text{ if } I \in D_i. \end{aligned}$$

The input system for obtaining an SSL-SD statement reordering function r consists of the following constraints:

$$\mathcal{C}_r = \{r_i(S_1) < r_j(S_2) \mid \text{there exist statements } S_1 \text{ and } S_2 \text{ in } \mathcal{S}, \text{ such that } S_1 \Rightarrow S_2 \text{ and } \mu_{SD}(g_t, S_1, S_2, i, j) = 0\}.$$

A Heuristic Algorithm

We use two heuristics to choose subdomain boundaries for SSL-SD schedules: using predicates in conditionals as subdomain boundaries, and using *inside-out* enumerative search to find subdomain boundaries.

Predicates in Conditionals Conditional of a statement specifies the subdomain of the index domain where the statement is active, and different conditionals specify different subdomains with different dependences. Therefore, predicates in conditionals may be a good guess for subdomain boundaries. And this appears to be the case for example programs including the dynamic programming example and the algebraic path problem to be discussed in Section 4.8.

Inside-Out Enumerative Search Another heuristic for generating subdomain boundaries is *inside-out* unbounded enumerative search [64,66]. Let a partitioning hyperplane be specified by $BI = c$, where B is a row vector, and c is a scalar. The inside-out enumerative search starts by setting the bounds of the absolute values of the elements of row vector B and scalar c to be 1, then 2, 3, etc., until a schedule is found, or a pre-determined bound is reached without obtaining a schedule successfully. Hopefully, if there exists a subdomain schedule, the values of the elements of B and c would be small. The largest absolute value of the elements of B and c is 2 for the dynamic programming example and the algebraic path problem to be discussed in Section 4.8.

Linear Programming Formulation Once the subdomains are determined, each polyhedron $Q(S_1, S_2, i, j)$ defined in Equation (4.38) can be decomposed into vertices and extremal rays. Therefore, the input system $\mathcal{C}_{g_t}$ defined in Equation (4.38) can be simplified to the following linear programming form:

$$\begin{aligned} \mathcal{C}_{g_t} = \{ & T_j \begin{bmatrix} V_J \\ 1 \end{bmatrix} - T_i \begin{bmatrix} V_I \\ 1 \end{bmatrix} \geq 0 \mid \text{for all vertex } (V_I, V_J) \text{ of } Q(S_1, S_2, i, j), \\ & \quad \text{where } S_1 \text{ and } S_2 \text{ are two statements in } \mathcal{S} \text{ such that } S_1 \Rightarrow S_2 \} \\ \cup \{ & T_j \begin{bmatrix} R_J \\ 0 \end{bmatrix} - T_i \begin{bmatrix} R_I \\ 0 \end{bmatrix} \geq 0 \mid \text{for all extremal ray } (R_I, R_J) \text{ of } Q(S_1, S_2, i, j), \\ & \quad \text{where } S_1 \text{ and } S_2 \text{ are two statements in } \mathcal{S} \text{ such that } S_1 \Rightarrow S_2 \}. \end{aligned} \tag{4.40}$$

We have the following theorem:

Theorem 4.1 Row vectors T_i and functions r_i, $1 \leq i \leq m$, define an SSL-SD schedule if and only if Equation (4.40) holds and $r_i(S_1) < r_j(S_2)$ is true whenever $\mu_{SD}(f, S_1, S_2, i, j) = 0$ holds.

Proof.

Only if Consider a dependence $S_1@I \Rightarrow S_2@J$ where $I \in D_i$ and $J \in D_j$. By the definition of a schedule, $T_j \begin{bmatrix} J \\ 1 \end{bmatrix} - T_i \begin{bmatrix} I \\ 1 \end{bmatrix} \geq 0$ must hold for all such index pairs

(I, J), including the vertex of $Q(S_1, S_2, i, j)$. On the other hand, if (V_I, V_J) is a vertex and (R_I, R_J) is an extremal ray of $Q(S_1, S_2, i, j)$, then index $(V_I + nR_I, V_J + nR_J)$ is also in $Q(S_1, S_2, i, j)$ for any positive integer n. By the definition of a schedule, $T_j \begin{bmatrix} V_J + nR_J \\ 1 \end{bmatrix} - T_i \begin{bmatrix} V_I + nR_I \\ 1 \end{bmatrix} \geq 0$ must hold for any positive integer n. That is, $T_j \begin{bmatrix} V_J \\ 1 \end{bmatrix} - T_i \begin{bmatrix} V_I \\ 1 \end{bmatrix} + n(T_j \begin{bmatrix} R_J \\ 0 \end{bmatrix} - T_i \begin{bmatrix} R_I \\ 0 \end{bmatrix}) \geq 0$ must hold for any positive integer n. This implies that $n(T_j \begin{bmatrix} R_J \\ 0 \end{bmatrix} - T_i \begin{bmatrix} R_I \\ 0 \end{bmatrix}) \geq 0$ must hold. Therefore, Equation (4.40) must hold for a valid schedule. Furthermore, when $\mu_{SD}(f, S_1, S_2, i, j) = 0$ holds, computation of $S_1@I$, $I \in D_i$, and computation of $S_2@J$, $J \in D_j$, can be scheduled in the same iteration of the transformed loop nest. Consequently, statement ordering $r_i(S_1)$ must be in front of $r_j(S_2)$ to satisfy the dependence.

If Consider a dependence $S_1@I \Rightarrow S_2@J$ where $I \in D_i$ and $J \in D_j$. Let $\mathcal{V}$ be the set of vertices and $\mathcal{R}$ be the set of extremal rays of $Q(S_1, S_2, i, j)$. Since (I, J) is the sum of a convex combination of the vertices (V_I, V_J) in $\mathcal{V}$ and of a positive combination of extremal rays (R_I, R_J) in $\mathcal{R}$, we have

$$(I, J) = \sum_x a_x (V_{Ix}, V_{Jx}) + \sum_y b_y (R_{Iy}, R_{Jy}), \text{ where } \sum_x a_x = 1,\ a_x \geq 0,\ b_y > 0.$$

Therefore,

$$T_j \begin{bmatrix} J \\ 1 \end{bmatrix} - T_i \begin{bmatrix} I \\ 1 \end{bmatrix} = \sum_x a_x (T_j \begin{bmatrix} V_{Jx} \\ 1 \end{bmatrix} - T_i \begin{bmatrix} V_{Ix} \\ 1 \end{bmatrix}) + \sum_y b_y (T_j \begin{bmatrix} R_{Jy} \\ 0 \end{bmatrix} - T_i \begin{bmatrix} R_{Iy} \\ 0 \end{bmatrix}).$$

Since Equation (4.40) holds, $T_j \begin{bmatrix} J \\ 1 \end{bmatrix} - T_i \begin{bmatrix} I \\ 1 \end{bmatrix} \geq 0$ also holds. In addition, if $T_j \begin{bmatrix} J \\ 1 \end{bmatrix} - T_i \begin{bmatrix} I \\ 1 \end{bmatrix}$ can be zero, then $r_i(S_1) < r_j(S_2)$ can preserve the dependence ordering. Therefore, The row vectors T_i and functions r_i, $1 \leq i \leq m$, define an SSL-SD schedule.
□

From Theorem 4.1, it is clear that, given subdomains of the index domain D of a loop nest, polyhedra decomposition, linear programming and topological sort can find an SSL-SD schedule if one exists. The dimension of the linear programming system is $m * (d + 1)$, where m is the number of subdomains and d is the dimension of the index domain of the loop nest.

A General Algorithm

We now present a general algorithm for finding SSL-SD schedules. Since the complexity of the general algorithm is unacceptably high, we will use the heuristic algorithm described above in practice. The reason for presenting the general algorithm is that it only requires bounded search space for subdomain boundaries so that it does not need to guess at subdomain boundaries as the heuristic does. However, nothing comes for free. The bounded search induces nonlinear programming, instead of linear programming, in finding an SSL-SD schedule.

In the following, we use a simple example to show why the search space can be bounded and why we need nonlinear programming. We then give a theorem about the complexity of the general method. The proof of the theorem is described in [77].

Bounded Search Space Let D be an unbounded rectangular domain $\{(i,j) \mid (i,j) \in [1,n] \times [1,n]\}$ where n is unknown at compile time. It is easy to see that polyhedron D has one vertex $(1,1)$ and two extremal rays $(0,1)$ and $(1,0)$. Suppose polyhedron D is separated by a vertical line $(i = a)$, where a is an unknown integer, into two subdomains $Q_1 = \{(i,j) \mid (i,j) \in D, i \leq a\}$ and $Q_2 = \{(i,j) \mid (i,j) \in D, i \geq a + 1\}$. We now need to find the vertices and extremal rays of Q_1 and Q_2. Since there can be infinite number of possible n and a, there are infinite number of possible sets of vertices and extremal rays. However, if we allow parameterized vertices and extremal rays, then there are only four possible sets of vertices and extremal rays of Q_1 and Q_2 as shown in Figure 4.4.

The four possible sets of vertices and extremal rays ofQ_1 and Q_2 are:

1. if $a < 1$: Q_1 is empty and Q_2 is D, as shown in Figure 4.4(a),

2. if $a = 1$: $(1,1)$ is a vertex and $(0,1)$ is an extremal ray of Q_1, and $(2,1)$ is a

vertex and $(0,1)$ and $(1,0)$ are extremal rays of Q_2, as shown in Figure 4.4(b),

3. if $n > a > 1$: $(1,1)$ and $(a,1)$ are vertices and $(0,1)$ is an extremal ray of Q_1, and $(a+1,1)$ is a vertex and $(0,1)$ and $(1,0)$ are extremal rays of Q_2, as shown in Figure 4.4(c),

4. if $a \geq n$: Q_1 is D and Q_2 is empty, as shown in Figure 4.4(d) and (e).

Note that vertices $(a,1)$ and $(a+1,1)$ in Figure 4.4(c) are parameterized using the unknown integer a. So the basic idea is that the search space of the subdomain boundaries and the vertices and extremal rays of those subdomains can be bounded if parameterized vertices and extremal rays are used.

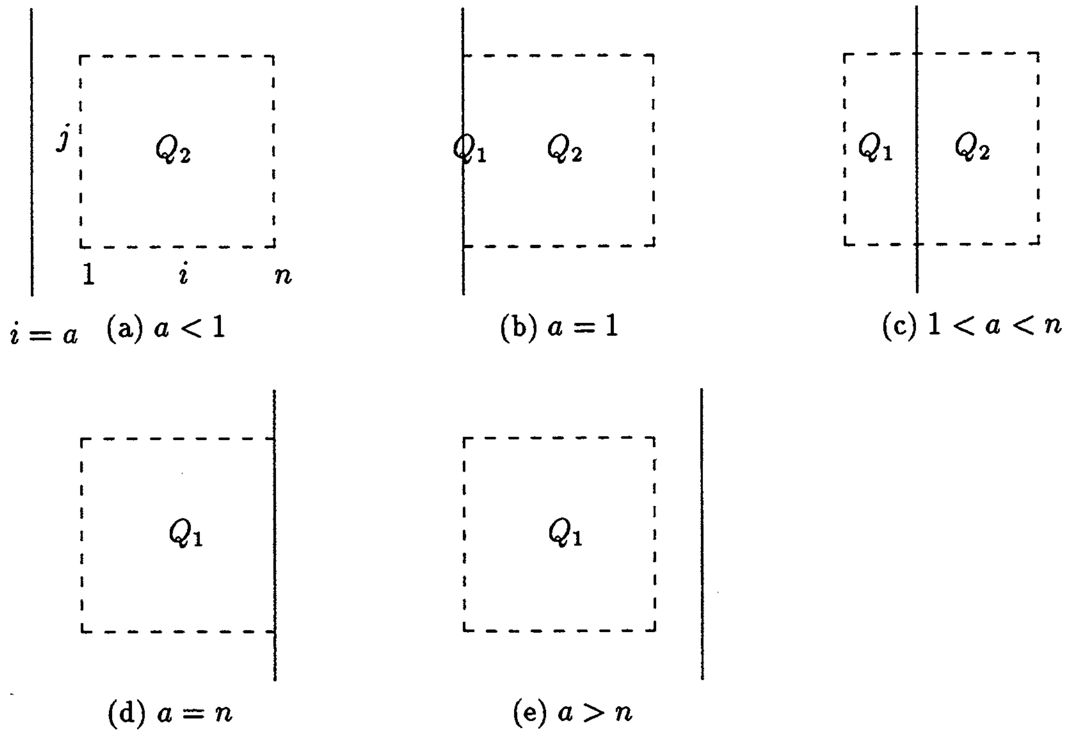

Figure 4.4: Bounded search space: an example.

Nonlinear Programming Recall that the input system defined in Equation (4.40) contains $T_j \begin{bmatrix} V_J \\ 1 \end{bmatrix} - T_i \begin{bmatrix} V_I \\ 1 \end{bmatrix} \geq 0$ and $T_j \begin{bmatrix} R_J \\ 0 \end{bmatrix} - T_i \begin{bmatrix} R_I \\ 0 \end{bmatrix} \geq 0$ as constraints. With unknown T_i, T_j, parameterized vertex (V_I, V_J) and parameterized extremal ray (R_I, R_J), the input system becomes a nonlinear one. Hence nonlinear programming is required to obtain the partitioning hyperplanes and the schedule for each disjoint subdomain.

We now give a theorem about the complexity of the general method for finding an SSL-SD schedule. The proof of the theorem is described in [77].

Theorem 4.2 An SSL-SD schedule with p partitioning hyperplanes on the domain can be obtained by enumerative search through a *bounded* search space over $\mathcal{B}$ where $\mathcal{B}$ contains all possible sets of parameterized polyhedra specifying dependences under p partitioning hyperplanes. At each instance of $\mathcal{B}$, nonlinear programming of order $2p + 1$ computes the partitioning hyperplanes and the schedule for each subdomain.

4.4.4 Statement-Variant Scheduling Algorithm

Single Statement in Each Partition From the subdomain schedule, we know that it is hard to find the partition of an index domain and the schedule for each subdomain at the same time. If the partition of the domain is given, then the problem becomes much simpler. This is also true for obtaining statement-variant schedules. Fortunately, for statement-variant schedules we can consider the extreme case where statements are partitioned into singleton sets. In this case, the problem reduces to subdomain scheduling problem with known subdomains. The input system $\mathcal{C}_{g_t}$ is:

$$\mathcal{C}_{g_t} = \{T_j \begin{bmatrix} J \\ 1 \end{bmatrix} - T_i \begin{bmatrix} I \\ 1 \end{bmatrix} \geq 0 \mid \text{ there exist statements } S_i \text{ and } S_j \text{ in } \mathcal{S}, \text{ such that } (I, J) \in \mathcal{P}(S_i, S_j)\}, \tag{4.41}$$

where $\mathcal{P}(S_i, S_j)$ is defined in Equation (4.37).

Linear Programming Formulation If we obtain the vertices and extremal rays of $\mathcal{P}(S_i, S_j)$, then the input system $\mathcal{C}_{g_t}$ defined in Equation (4.41) can be simplified to:

$$\begin{aligned}
\mathcal{C}_{g_t} = \{T_j \begin{bmatrix} V_J \\ 1 \end{bmatrix} \; -T_i \begin{bmatrix} V_I \\ 1 \end{bmatrix} &\geq 0 \mid \text{for all vertex } (V_I, V_J) \text{ of } \mathcal{P}(S_i, S_j), \\
&\text{where } S_i \text{ and } S_j \text{ are two statements in } \mathcal{S} \text{ such that } S_i \Rightarrow S_j\} \\
\cup \; \{T_j \begin{bmatrix} R_J \\ 0 \end{bmatrix} \; -T_i \begin{bmatrix} R_I \\ 0 \end{bmatrix} &\geq 0 \mid \text{for all extremal ray } (R_I, R_J) \text{ of } \mathcal{P}(S_i, S_j), \\
&\text{where } S_i \text{ and } S_j \text{ are two statements in } \mathcal{S} \text{ such that } S_i \Rightarrow S_j\}.
\end{aligned} \tag{4.42}$$

Statement Reordering Let Γ, E_t, $\mathcal{Z}$ be as defined before. We define a second-order function $\mu_{SV}(f, S_1, S_2)$ to be the minimal target difference vector ranging over the vectors $f(J) - f(I)$ where $(I, J) \in \mathcal{P}(S_1, S_2)$ and $f \in \Gamma$:

$$\begin{aligned}
\mu_{SV} \; &: \; \Gamma \times \mathcal{S} \times \mathcal{S} \to \mathcal{Z} \\
\mu_{SV}(f, S_i, S_j) \; &= \; \min\{f(J) - f(I) \mid (I, J) \in \mathcal{P}(S_1, S_2)\} \\
f(I) \; &= \; T_i \begin{bmatrix} I \\ 1 \end{bmatrix} = g_t(I) \;\; \text{for the computation of} \;\; S_i@I.
\end{aligned} \tag{4.43}$$

The input system for obtaining an SSL-SV statement reordering function r consists of the following constraints:

$$\begin{aligned}
\mathcal{C}_r = \{r(S_1) < r(S_2) \mid \; &\text{there exist statements } S_1 \text{ and } S_2 \text{ in } \mathcal{S}, \\
&\text{such that } S_1 \Rightarrow S_2 \text{ and } \mu_{SV}(g_t, S_1, S_2) = 0\}.
\end{aligned}$$

We have the following theorem:

Theorem 4.3 Row vectors T_i, $1 \leq i \leq s$, and a function r define an SSL-SV schedule if and only if Equation (4.42) holds and $r(S_i) < r(S_j)$ is true whenever $\mu_{SV}(f, S_i, S_j) = 0$ holds.

The proof of this theorem is omitted since it is very similar to the proof of Theorem 4.1.

From the above theorem, it is clear that polyhedra decomposition, linear programming and topological sort can find an SSL-SV schedule if one exists. The dimension of the linear programming system is $s * (d + 1)$, where s is the number of statements and d is the dimension of the index domain of the loop nest.

Algorithm for Generating Nonuniform Schedules Since a nonuniform schedule is a combination of a subdomain schedule and a statement-variant schedule, it can be obtained by the algorithms described in Sections 4.4.3 and 4.4.4.

4.5 An Iterative Algorithm for Generating Multiple-Sequential Level Schedules

We now discuss extending single-sequential level scheduling algorithms to multiple-sequential level and mixed scheduling algorithms. As stated before, we assume the target parallel machines are fine-grain. Therefore, all parallel loops are innermost loops in the transformed loop nest. We start with multiple-sequential level scheduling algorithm.

To obtain an n-sequential level schedule, our approach is to first find n SSL temporal morphisms g_t^1, ..., g_t^n, one at a time, followed by generating the statement reordering function r. This technique applies to all schedules: uniform, subdomain-variant, statement-variant or nonuniform.

To describe an iterative algorithm, we need to define a sequence of minimal target difference vectors similar to the ones defined in Equations (4.31), (4.39) and (4.43).

A Sequence of Target Difference Vectors We first define Γ_i, $1 \leq i \leq n$, to be the function space $[D \rightarrow E_t^1 \times \ldots \times E_t^i]$, where each E_t^j, $1 \leq j \leq n$, is a one-dimensional temporal index domain, and define $\mathcal{Z}$ to be the set of rationals.

For uniform schedules, we define a family of second-order functions $\mu_i(f, S_1, S_2)$, $1 \leq i \leq d$, to be a sequence of minimal target difference vectors (in the sense of lexicographical ordering on elements of $E_t \times \ldots \times E_t^i$) ranging over the images of the set of dependence vectors $\mathcal{V}(S_1, S_2)$ defined in Equation (4.26) under function $f \in \Gamma_i$:

$$\begin{aligned} \mu_i &: \Gamma_i \times \mathcal{S} \times \mathcal{S} \rightarrow \mathcal{Z}, \text{ where} \\ \mu_i(f, S_1, S_2) &= \min\{f(K) \mid K \in \mathcal{V}(S_1, S_2)\}. \end{aligned} \tag{4.44}$$

Similarly, a family of μ functions can be defined for subdomain, statement-variant and nonuniform schedules.

Iterative Algorithm Similar to the algorithm for obtaining SSL schedules with statement reordering functions as discussed in Section 4.4.2, the algorithm for generating multiple-sequential level schedules starts with criterion $\mu(g_t^1, S_1, S_2) \geq 0$ for all pairs of statements S_1 and S_2 such that $S_1 \Rightarrow S_2$ to generate an SSL temporal morphism g_t^1. We then use the constraint $\mu(g_t^2, S_1, S_2) \geq 0$ for those dependences $S_1 \Rightarrow S_2$ such that $\mu(g_t^1, S_1, S_2) = 0$ to find g_t^2. The same process is iterated until the n-th iteration, $1 \leq n \leq d$, when the remaining dependences $S_1 \Rightarrow S_2$ where $\mu_n((g_t^1, \ldots, g_t^n), S_1, S_2) = \hat{0}$ are not cyclic. In this case, the set of statements $\mathcal{S}$ has the partial ordering defined in Section 4.4.2. We then use topological sort to find the linear ordering of the statements.

We summarize the above discussion as the following iterative algorithm for generating multiple-sequential level schedules: The inputs of Algorithm MSL consist of the set of all dependences in a loop nest, denoted by $\mathcal{L}$, and the choice of one of the algorithms presented in Section 4.4.1, 4.4.3 or 4.4.4, denoted by $\mathcal{A}$.

```
Algorithm MSL ($\mathcal{L}$ : a set of dependences, $\mathcal{A}$ : an algorithm)
  1. $i \leftarrow 0$ ($i$ will be ranging over the loop levels);
  2. While $i \leq d$ and $\mathcal{L}$ contains cyclic dependences, do
      (a) $i \leftarrow i + 1$;
```

   (b) Find an SSL temporal morphism g_t^i by using algorithm $\mathcal{A}$ such that $\mu(g_t^i, S_1, S_2) \geq 0$ holds for all $S_1 \Rightarrow S_2 \in \mathcal{L}$;

   (c) If such g_t^i does not exist, then the algorithm terminates without returning a schedule;

   (d) Else, $\mathcal{L} \leftarrow \{(S_1 \Rightarrow S_2) \mid (S_1 \Rightarrow S_2) \in \mathcal{L}$ such that $\mu(g_t^i, S_1, S_2) = 0\}$;

3. (Now $\mathcal{L}$ contains no cyclic dependences.) Find a statement reordering function r by topological sort, such that $r(S_1) < r(S_2)$ for all $(S_1 \Rightarrow S_2) \in \mathcal{L}$.

Note that the sequential loops in the transformed loop nest generated by Algorithm MSL are perfectly nested.

4.6 A Recursive Algorithm for Generating Mixed Schedules

We now present the algorithm for generating mixed schedules. We first discuss the basic idea of this algorithm.

Basic Idea Algorithm MSL presented in the previous section for generating multiple-sequential level schedules treats all the statements in the same way throughout the iterations even though more and more of the dependences become uninformative in obtaining the sequence $g_t^1, \ldots, g_t^n$. Clearly, when some instances of dependence relations are not considered, a new set of equivalence classes under relation "$\sim$" over statements emerges. As discussed in Chapter 2, these new equivalence classes can be scheduled separately for the subsequent iterations. To do this, a tree of temporal morphisms, instead of just a sequence $g_t^1, \ldots, g_t^n$, will be generated.

Labeling a Tree We use a prefix notation to label each node of the trees of temporal morphisms, equivalence classes and sets of dependence relations to be defined later.

Let the root of the tree be labeled 1. For a node which is the i-th child of its parent node, we say the *order* of this node is i. A node is labeled $l \circ i$ if l is the label of its parent node and it has order i. A temporal morphism g_t with label l is denoted by $g_t(l)$, an equivalence class B with label l is denoted by $\mathcal{S}(l)$, and a set of dependences $\mathcal{L}$ with label l is denoted by $\mathcal{L}(l)$.

Recursive Algorithm The initial inputs of the recursive algorithm for generating mixed schedules consist of the equivalence class $\mathcal{S}(1)$, which contains all statements in $\mathcal{S}$, and the set $\mathcal{L}(1)$, which contains all dependences in the source loop nest. With inputs $\mathcal{S}(l)$ and $\mathcal{L}(l)$, the algorithm obtains an SSL temporal morphism $g_t(l)$ such that $\mu(g_t(l), S_1, S_2) \geq 0$ holds for all $(S_1 \Rightarrow S_2) \in \mathcal{L}(l)$. Under $g_t(l)$, the set of dependences

$$\mathcal{U}(l) = \{(S_1 \Rightarrow S_2) \mid (S_1 \Rightarrow S_2) \in \mathcal{L}(l), \mu(g_t(l), S_1, S_2) > 0\} \tag{4.45}$$

becomes uninformative and should be removed from $\mathcal{L}(l)$. A new set of equivalence classes, denoted by $\mathsf{new}(l)$, emerges:

$$\mathsf{new}(l) = \{\mathcal{S}(l \circ i) \mid \mathcal{S}(l \circ i) \text{ is the } i\text{-th equivalence class from } \mathcal{L}(l) - \mathcal{U}(l)\}. \tag{4.46}$$

For each new equivalence class $\mathcal{S}(l \circ i)$, $1 \leq i \leq |\mathsf{new}(l)|$, the associated set of dependences $\mathcal{L}(l \circ i)$ is

$$\mathcal{L}(l \circ i) = \{(S_1 \Rightarrow S_2) \mid S_1 \in \mathcal{S}(l \circ i), S_2 \in \mathcal{S}(l \circ i), (S_1 \Rightarrow S_2) \in (\mathcal{L}(l) - \mathcal{U}(l))\} \tag{4.47}$$

If $\mathcal{S}(l \circ i)$, $1 \leq i \leq |\mathsf{new}(l)|$, is a dependent block, then the same algorithm is applied recursively to the new inputs $\mathcal{S}(l \circ i)$ and $\mathcal{L}(l \circ i)$. The recursive algorithm stops when all dependent blocks have been broken up and all remaining blocks are independent.

We summarize the above discussion as the following recursive algorithm for generating mixed schedules: The initial inputs of Algorithm MIX consist of the equivalence class $\mathcal{S}(1)$, the set of dependences $\mathcal{L}(1)$, and the choice of one of the algorithms presented in Section 4.4.1, 4.4.3 or 4.4.4, denoted by $\mathcal{A}$.

```
Algorithm MIX ($\mathcal{S}(l)$ : an equivalence class, $\mathcal{L}(l)$ : a set of dependences,
  $\mathcal{A}$ : an algorithm)
```

1. Find an SSL temporal morphism $g_t(l)$ by using algorithm $\mathcal{A}$, such that $\mu(g_t(l), S_1, S_2) \geq 0$ holds for all $(S_1 \Rightarrow S_2) \in \mathcal{L}(l)$;
2. If such $g_t(l)$ does not exist, then the algorithm terminates without returning a schedule.
3. obtain the set $\mathsf{new}(l)$ (remove uninformative dependences and generate new equivalence classes);
4. Generate new sets of dependences $\mathcal{L}(l \circ i)$, $1 \leq i \leq |\mathsf{new}(l)|$;
5. For $i \in [1, |\mathsf{new}(l)|]$, if $\mathcal{S}(l \circ i)$ is a dependent block, then MIX($\mathcal{S}(l \circ i), \mathcal{L}(l \circ i), \mathcal{A}$).

Statement Reordering We now discuss how to obtain the statement reordering function r for mixed schedules. Since the algorithm for generating mixed schedules is recursive, statement ordering is also obtained recursively. Suppose we want to find the ordering among the statements in an equivalence class $\mathcal{S}(l)$. Let $\mathcal{S}(l \circ i)$, $1 \leq i \leq n$, be the equivalence classes in $\mathsf{new}(l)$ generated from $\mathcal{S}(l)$. We can first determine the ordering among these n equivalence classes, which induces the ordering among statements in different $\mathcal{S}(l \circ i)$, but not the ordering among statements within the same $\mathcal{S}(l \circ i)$. Since each $\mathcal{S}(l \circ i)$, $1 \leq i \leq n$, will be scheduled separately, the ordering among statements in different $\mathcal{S}(l \circ i)$ will never be changed subsequently. The same process is then applied recursively to each $\mathcal{S}(l \circ i)$, $1 \leq i \leq n$, to determine the ordering among statements within $\mathcal{S}(l \circ i)$.

The ordering among these n equivalence classes is obtained as follows. Similar to Section 4.4.2, we define a partial ordering "$<$" over the set of equivalence classes $\mathsf{new}(l)$. We say $\mathcal{S}(l \circ 1) < \mathcal{S}(l \circ 2)$, i.e. equivalence class $\mathcal{S}(l \circ 1)$ must be in front of equivalence class $\mathcal{S}(l \circ 2)$ in the transformed loop body, if there exist statements $S_1 \in \mathcal{S}(l \circ 1)$ and $S_2 \in \mathcal{S}(l \circ 2)$ such that $(S_1 \Rightarrow S_2) \in \mathcal{L}(l)$ and $\mu(g_t(l), S_1, S_2) = 0$.

Due to the way these n equivalence classes are generated from $\mathcal{S}(l)$, $\mathsf{new}(l)$ always has this partial ordering. Therefore, topological sort can produce the linear ordering of these n equivalence classes.

Imperfect Loop Nests For a given statement S, it may belong to a nested level of dependent blocks $\mathcal{S}(1)$, ..., $\mathcal{S}(l)$, where $\mathcal{S}(1) \supset \ldots \supset \mathcal{S}(l)$. Clearly, the schedule of statement S is $|l|$-sequential level, where $|l|$ is the length of the label l:

$$\pi(I,S) = (g_t(1)(I), \ldots, g_t(l)(I), r(S)). \tag{4.48}$$

Since different statements can belong to different sets of dependent blocks, the sequential level of $\pi(I,S)$ can be different for different statements. And the sequential loops in the transformed parallel loop nest can be of any form, i.e. perfectly and imperfectly nested loops.

When different equivalence classes are scheduled differently, each SSL temporal morphism will be a statement-variant schedule with the partition of statements being these new equivalence classes. Recall that in Section 4.4.4, a statement-variant schedule is obtained by allowing each statement to be scheduled differently, but not independently. The advantage to schedule independently is that the likelihood of obtaining a suitable schedule with more parallelism increases if each dependent block is considered separately.

An Example We use the following example to explain Algorithm MIX further. In this example, each SSL temporal morphism used in Algorithm MIX is a uniform one.

Loop Nest 4.16

```
DO ($i = 2, n$) {
    DO ($j = 2, n$) {
        DO ($k = 2, n$) {
            $S_1 : A(i,j,k) = A(i-1,k,j) + B(i-1,j,k)$
            $S_2 : B(i,j,k) = B(i,j-1,j) + C(i-1,j,k)$
            $S_3 : C(i,j,k) = C(i,j,k-1) + A(i,j,k)$    } } }
```

The initial $\mathcal{S}(1)$ is the set $\{S_1, S_2, S_3\}$ and $\mathcal{L}(1)$ is the set $\{(S_x \Rightarrow S_y) \mid (x,y) \in \{(1,1),(2,2),(3,3),(2,1),(3,2),(1,3)\}$. It is easy to check that if $g_t(1)((i,j,k),S) = i$ for all $S \in \mathcal{S}(1)$, then $\mu(g_t(1), S_x, S_y) = 1$ for (x,y) in the set $U = \{(1,1),(2,1),(3,2)\}$. And $\mu(g_t(1), S_x, S_y) = 0$ for (x,y) in $(x,y) \in \{(2,2),(3,3),(1,3)\}$. Consequently, dependences $S_x \Rightarrow S_y$, $(x,y) \in U$, can be removed and $\mathcal{S}(1)$ is broken up into three equivalence classes $\mathcal{S}(1 \circ i)$, $1 \leq i \leq 3$; each contains a single statement S_i.

Since $\mathcal{S}(1 \circ 1)$ is an independent block, the schedule for S_1 will be single-sequential level. Furthermore, S_2 and S_3 are in different equivalence classes and they can be scheduled independently. Let $g_t(1 \circ 2)((i,j,k),S) = j$ and $g_t(1 \circ 3)((i,j,k),S) = k$. It is easy to check that $\mu(g_t(1 \circ 2), S_2, S_2) = 1$ and $\mu(g_t(1 \circ 3), S_3, S_3) = 1$.

To summarize, the mixed schedule is single-sequential level for S_1 (two-dimensional parallelism), and is two-sequential level for S_2 and S_3 (one-dimensional parallelism) as shown below:

$$\pi((i,j,k),S) = \left\{ \begin{array}{l} S = S_1 \rightarrow (i,1) \\ S = S_2 \rightarrow (i,j,2) \\ S = S_3 \rightarrow (i,k,3) \end{array} \right\}. \tag{4.49}$$

Because $\mu(g_t(1), S_1, S_3) = 0$ and statements S_1 and S_3 are in different equivalence classes, S_1 must be in front of S_3 in the transformed loop body. The resulting parallelized program with imperfectly nested sequential loops is:

Loop Nest 4.17

```
DO $(i = 2, n)$ {
  DOALL $((j = 2, n), (k = 2, n))$ {
    $S_1: A(i, j, k) = A(i-1, k, j) + B(i-1, j, k)$ }
  DO $(j = 2, n)$ {
    DOALL $(k = 2, n)$ {
      $S_2: B(i, j, k) = B(i, j-1, j) + C(i-1, j, k)$ } }
  DO $(k = 2, n)$ {
    DOALL $(j = 2, n)$ {
      $S_3: C(i, j, k) = C(i, j, k-1) + A(i-1, j, k)$ } } }
```

4.7 The Structure of Static Scheduler

Figure 4.5 shows the structure of a compiler for parallelizing affine loops (defined in Section 4.1.1). It includes the subdomain dependence test presented in Chapter 3, new loop transformations presented in this chapter, and related previous techniques.

As discussed in Chapter 1, the compilation process in Figure 4.5 consists of three major phases: static parallelism detection, data layout and communication analysis, and code generation. We now discuss the static scheduling part in more detail.

Although the subdomain dependence test is more accurate for statements with conditionals and with coupled array subscripts, its complexity is higher than some previous dependence tests, e.g. the GCD test [6]. Therefore, the GCD test and other fast dependence tests can be used first to filter out some independent computations. The subdomain dependence test can then be used between the remaining dependent statements to report more accurate dependences.

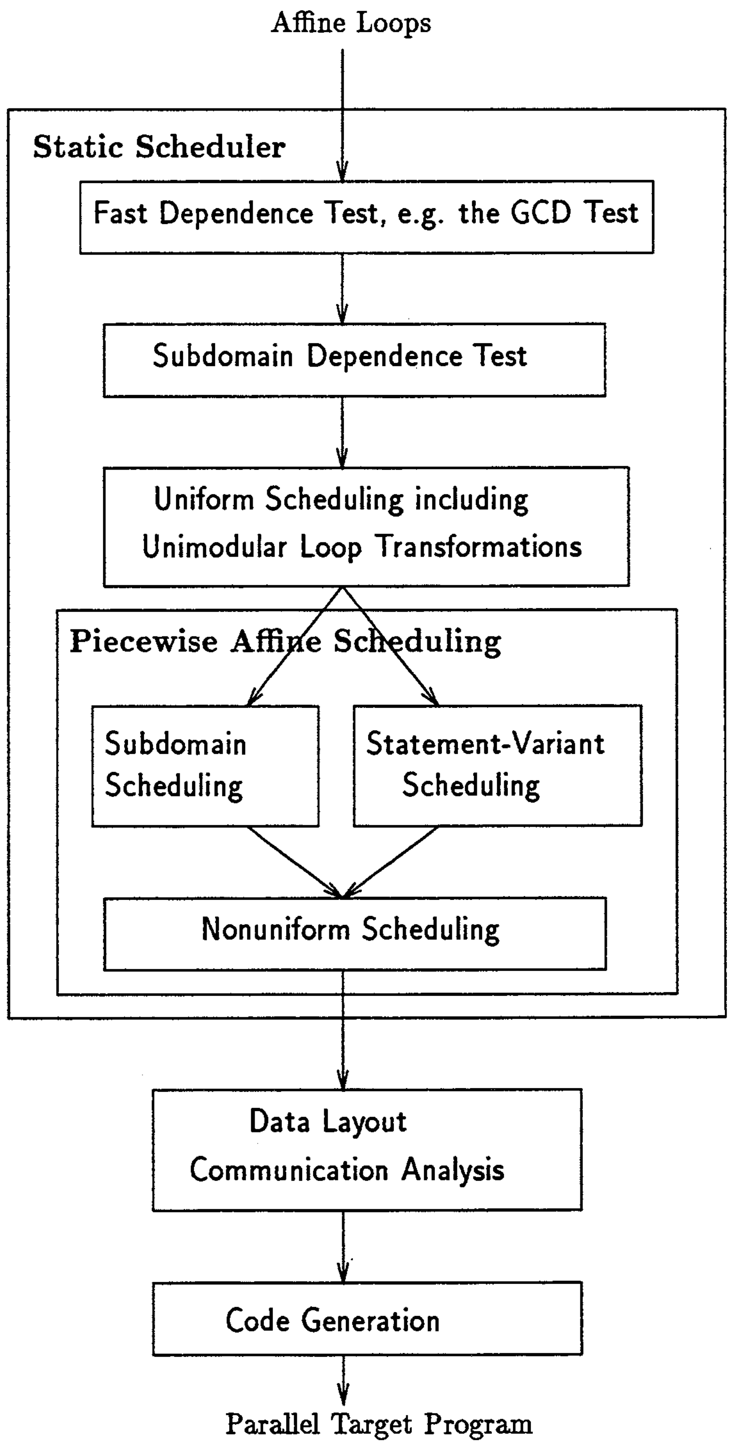
Affine Loops
Static Scheduler
Fast Dependence Test, e.g. the GCD Test
Subdomain Dependence Test
Uniform Scheduling including Unimodular Loop Transformations
Piecewise Affine Scheduling
Subdomain Scheduling
Statement-Variant Scheduling
Nonuniform Scheduling
Data Layout Communication Analysis
Code Generation
Parallel Target Program

Figure 4.5: The structure of the static parallelization system.

On the loop transformation part, piecewise affine scheduling algorithms can detect more parallelism than uniform scheduling method (including unimodular loop transformations [10]), but their complexity is also higher. Hence, piecewise affine scheduling algorithms are used when the uniform scheduling method cannot reveal enough parallelism. Among piecewise affine schedules, nonuniform scheduling algorithm is more powerful and more complex than subdomain and statement-variant scheduling algorithms, but it is noncomparable between the latter two. Therefore, subdomain and statement-variant scheduling algorithms are used after uniform scheduling algorithm, and nonuniform scheduling algorithm comes last as shown in Figure 4.5.

In addition to the ordering between using uniform, subdomain, statement-variant and nonuniform scheduling methods, there is also an ordering between using single-sequential level, multiple-sequential level, and mixed scheduling algorithms. Since a single-sequential level schedule has maximal degree of parallelism, the single-sequential level scheduling method is used first. If it fails to find a single-sequential level schedule, then the multiple-sequential level scheduling method is used next. The mixed scheduling method comes last when the obtained multiple-sequential level schedule does not posses enough parallelism.

To summarize, we begin with the single-sequential level uniform scheduling method, and follow by the multiple-sequential level uniform scheduling method. Since there is no mixed uniform schedule, the next one we try is single-sequential level subdomain or statement-variant scheduling method. The scheduling process stops when enough parallelism is detected or the mixed nonuniform scheduling method, the last one in our classification, is used, which reveals most parallelism among these techniques.

The subdomain dependence test and previous dependence analysis can provide dependence information and the corresponding direction vectors between statements. Direction vectors are very useful for loop transformations such as loop vectorization, reversal, interchange and permutation. For piecewise affine loop transformations, the notion of dependence index pair (defined in Section 4.4.3) is necessary to capture more precise dependence information. Since the complexity of polyhedra decomposition, which can obtain representative dependence index pairs, is NP-complete [35,100], we had better avoid decomposing empty polyhedra resulting from independent computations. That is, efficient dependence analysis can still be used to detect empty polyhedra before exercising polyhedra decomposition. Therefore, dependence analysis can provide information to (1) determine whether loop transformations such as loop vectorization, reversal, interchange and permutation are applicable, and (2) detect empty polyhedra to save the compilation time for obtaining piecewise affine loop transformations.

4.8 Applications of Subdomain and Statement-Variant Schedules

To illustrate the usefulness of the new scheduling algorithms, we take dynamic programming, transitive closure and shortest path problems as examples.

4.8.1 Dynamic Programming

Source Code Dynamic programming has sequential complexity $O(n^3)$ for a problem of size n. The source code is given in Loop Nest 4.18.

Loop Nest 4.18

```
DO ($i = 1, n-2$) {
  DO ($j = i+2, n$) {
    $C(i,j) = \mathsf{min}_{i<k<j}(h(C(i,k), C(k,j)))$ } }
```

This source program is first transformed to the following form in a systematic manner by applying *fan-in* and *fan-out* reductions [17] to reduce potential concurrent accesses of variables. The result is Loop Nest 4.19:

Loop Nest 4.19

```
DO $(i = n-1, 1, -1)$ {
    DO $(j = i+1, n)$ {
        DO $(k = i, j)$ {
            $S_{a1}$ : IF$(k < j)\, A(i,j,k) = A(i,j-1,k)$
            $S_{b1}$ : IF$(i+1 = k)\, B(i,j,k) = C(i+1,j,j)$
            $S_{b2}$ : IF$(i+1 < k)\, B(i,j,k) = B(i+1,j,k)$
            $S_{c1}$ : IF$(i+j+1 = 2k)\, C(i,j,k) = h_1(A(i,j,k), B(i,j,k),$
                $A(i,j,i+j-k), B(i,j,i+j-k))$
            $S_{c2}$ : IF$(i+j+1 < 2k < 2j)\, C(i,j,k) = h_2(C(i,j,k-1), A(i,j,k),$
                $B(i,j,k), A(i,j,i+j-k), B(i,j,i+j-k))$
            $S_{c3}$ : IF$(k = j)\, C(i,j,k) = C(i,j,k-1)$
            $S_{a2}$ : IF$(k = j)\, A(i,j,k) = C(i,j,k)$                    } } }
```

Schedules We wrote three *Lisp programs on the Connection Machine CM/2, each with the control structure generated by a two-sequential level uniform schedule, a mixed statement-variant schedule and a single-sequential level subdomain schedule respectively. These schedules are generated according to the algorithms described before. For the subdomain schedule, the subdomain boundary $(i + j + 1 = 2k)$ is chosen from the predicate of statement S_{c1}. We also have a sequential Common-Lisp program on the Symbolics to compute the same problem. The three schedules are

n	3-sequential level sequential	2-sequential level uniform	mixed statement-variant	1-sequential level subdomain
32	6.8	10.72	2.47	0.87
64	55.0	42.88	9.73	1.73
128	440.0	171.50	39.16	3.48
256	3520.0	686.45	235.70	6.96
512	28160.0	2745.80	1159.24	31.70

Table 4.2: Running time in seconds of dynamic programming on an 8K-processor CM/2.

given below. For simplicity, we do not give the constant terms $r(S)$ of function π.

two-sequential level uniform schedule:

$$\pi((i,j,k),S) = (j-i, k-i)$$

mixed statement-variant schedule:

$$\pi(S,(i,j,k)) = \left\{ \begin{array}{l} S = S_{c2} \rightarrow (j-i, k-i) \\ \text{else} \rightarrow (j-i) \end{array} \right\}$$

single-sequential level subdomain schedule:

$$\pi(S,(i,j,k)) = \left\{ \begin{array}{l} i+j+1 \leq 2k \rightarrow -2i+j+k \\ i+j+1 > 2k \rightarrow -i+2j-k \end{array} \right\}.$$

Recall that Loop Nest 4.12 is part of Loop Nest 4.19. Therefore, the transformed Loop Nest 4.13, 4.14 and 4.15 are part of the transformed loop nests of the dynamic programming source code for these three schedules respectively.

Experimental Result The experiment is conducted as follows: we run the sequential code on the Symbolics and parallel codes on an 8K-processor Connection Machine with Symbolics as its host. The results described in Table 4.2 and Figure 4.6 show that

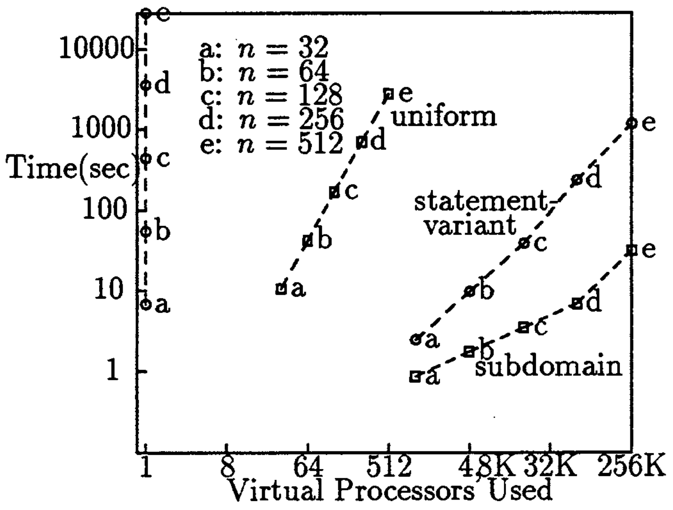

Figure 4.6: Running time vs. problem size.

the version using a single-sequential level subdomain schedule is three orders of magnitude faster than the sequential code, and is two orders of magnitude faster than the versions using a two-sequential level uniform schedule and mixed statement-variant schedule. And the program using a mixed statement-variant schedule is about three to four times faster than the program using a two-sequential level uniform schedule.

4.8.2 Transitive Closure and Shortest Path Problems

The Warshall's algorithm for the transitive closure problem and the Floyd's algorithm for the shortest path problem can be expressed in Loop Nest 4.20 with sequential complexity $O(n^3)$ for a problem of size n.

Loop Nest 4.20

```
DO $(k = 1, n)\{$
  DO $(i = 1, n)\{$
    DO $(j = i, n)\{$
      $X(i,j,k) = X(i,j,k-1) + X(i,k,k-1) * X(k,j,k-1)\ \}\ \}\ \}$
```

This source program can be transformed to the following form by adding propagating variables R and C [60,75]:

Loop Nest 4.21

```
DO ($k = 1, n$) {
    DO ($i = 1, n$) {
        DO ($j = i, n$) {
            $S_1$ : IF($j = k$) $C(i,j,k) = X(i,j,k-1)$
            $S_2$ : IF($j < k$) $C(i,j,k) = C(i,j+1,k)$
            $S_3$ : IF($j > k$) $C(i,j,k) = C(i,j-1,k)$
            $S_4$ : IF($i = k$) $R(i,j,k) = X(i,j,k-1)$
            $S_5$ : IF($i < k$) $R(i,j,k) = R(i+1,j,k)$
            $S_6$ : IF($i > k$) $R(i,j,k) = R(i-1,j,k)$
            $S_7$ : $X(i,j,k) = X(i,j,k-1) + R(i,j,k) * C(i,j,k)$ } } }
```

Schedule A single-sequential level subdomain schedule can be generated according to the algorithms described before. Two subdomain boundaries $(i = k)$ and $(j = k)$ are chosen from predicates. The generated subdomain schedule is

$$\pi((i,j,k),S) = \begin{cases} (j \geq k) \wedge (i \geq k) \rightarrow (i+j+k, r_1(S)) \\ (j < k) \wedge (i \geq k) \rightarrow (i-j+3k, r_2(S)) \\ (j \geq k) \wedge (i < k) \rightarrow (-i+j+3k, r_3(S)) \\ (j < k) \wedge (i < k) \rightarrow (-i-j+5k, r_4(S)), \end{cases}$$

where $r_i(S) = 2$ for $S = S_7$, $1 \leq i \leq 4$, and $r_i(S) = 1$ for other statements, $1 \leq i \leq 4$. This schedule with spatial morphism $g_s(i,j,k) = (i,j)$ transforms Loop Nest 4.21 into Loop Nest 4.22 below. The transformed loop nest has $(5n-4)$ sequential steps, which is the same as those presented in [60] and [75]. However, our scheduling algorithm is systematic while their schedules were obtained by ad hoc methods.

Loop Nest 4.22

```
DO ($t = 3, 5n - 2$) {
  DOALL ($i = 1, n$) {
    DOALL ($j = i, n$) {
      $k = t - i - j$
      IF($(1 \leq k \leq n) \wedge (j \geq k) \wedge (i \geq k)$) $S_1$, $S_2$, $S_3$, $S_4$, $S_5$, $S_6$
      $k = (t - i + j)/3$
      IF($(1 \leq k \leq n) \wedge (j < k) \wedge (i \geq k)$) $S_1$, $S_2$, $S_3$, $S_4$, $S_5$, $S_6$
      $k = (t + i - j)/3$
      IF($(1 \leq k \leq n) \wedge (j \geq k) \wedge (i < k)$) $S_1$, $S_2$, $S_3$, $S_4$, $S_5$, $S_6$
      $k = (t + i + j)/5$
      IF($(1 \leq k \leq n) \wedge (j > k) \wedge (i > k)$) $S_1$, $S_2$, $S_3$, $S_4$, $S_5$, $S_6$
      $k = t - i - j$
      IF($(1 \leq k \leq n) \wedge (j \geq k) \wedge (i \geq k)$) $S_7$
      $k = (t - i + j)/3$
      IF($(1 \leq k \leq n) \wedge (j < k) \wedge (i \geq k)$) $S_7$
      $k = (t + i - j)/3$
      IF($(1 \leq k \leq n) \wedge (j \geq k) \wedge (i < k)$) $S_7$
      $k = (t + i + j)/5$
      IF($(1 \leq k \leq n) \wedge (j > k) \wedge (i > k)$) $S_7$                }}}
```

Chapter 5

Dynamic Scheduler Generator

5.1 Motivation

Automatic parallelization has so far been unsuccessful in dealing with many real-world programs where extensive indirect array references or pointers are used. Though programs using pointers can be analyzed to some extent [16,42,44,46,49,62,63], those containing input-dependent or dynamic-changing structures are not amenable to compile-time analysis. For such programs, many run-time scheduling techniques have been proposed [26,27,33,80,82,87,95,97,98,101,104,115,114,116].

In this chapter, we present a hybrid compile-time and run-time approach where a *scheduler* is generated by the compiler based on information deduced from static analysis. At run-time, the scheduler records dynamic data references and allocates work to processors based on the run-time reference patterns. The central point is that compiler analysis can help to make the overhead incurred by the run-time scheduler insignificant, thereby lessening the major problem often faced by a run-time system.

In this context, the compiler's main task is to do *scheduler generation* in addition to *parallel program generation.* Upon seeing each read (or write) reference, the *scheduler generator* emits a call to a *recording procedure* for flow dependence (or anti- or output dependence) in conjunction with the necessary *static* program slices [105] that control the program's flow to that reference. However, care must be taken to avoid either the space or time blowing up due to the sheer number of references in any real program. Those indirect references or pointers that do not produce dependences (e.g. pointer dereferencing, induction variable, etc.), or produce only redundant dependences, must be identified and no calls to the recording procedure shall be emitted for them. The technique is called *redundant reference elimination* (RRE).

5.1.1 Application Domain: Restricted Nonaffine Loops

Our approach applies to both FORTRAN-style programs with indirect array references and C-style programs with pointers. In contrast to the affine loops defined in Section 4.1.1, we call loops with indirections or pointers *nonaffine loops.* We find that compiler analysis can make the run-time scheduling overhead insignificant for loops in a subclass of nonaffine loops. We call the loops in this subclass *restricted nonaffine loops.* Since defining restricted affine loops requires some terminologies presented below, the definition will be given in Section 5.5.1.

Organization of the Chapter The rest of the chapter is organized as follows. In Section 5.2, we survey various run-time parallelization approaches by classifying them according to (1) how and when work is assigned to processors, (2) how the work is scheduled within each processor. We also compare our work with related work, in both run-time scheduling and redundant dependence elimination. In Section 5.3, we first give an overview of our system, and then present the main technique of scheduler generation with algorithms for the recording procedures. In Section 5.4, we describe several redundant reference elimination techniques. Finally, we illustrate the usefulness of the dynamic scheduler generator with example programs in Section 5.5.

5.2 A Survey of Run-time Loop Parallelization Approaches

5.2.1 Characterizing Loop-Parallelization Techniques

Various run-time loop parallelization (automatic or manual) approaches can be broadly characterized by two orthogonal factors: (1) *assignment strategy* specifying how and when work is assigned to processors and (2) *scheduling method* dictating how the work is scheduled within each processor. We discuss three assignment strategies and two scheduling methods as follows:

Assignment Strategies

Compile-time: The work is assigned at compile-time, independent of any input data or dynamic behavior of the program.

Run-time Invariant: The work is assigned at run-time but before entering the loop, and stays invariant throughout the entire loop execution.

Run-time Dynamic: The work is assigned to processors during the loop execution, and the behavior of the computation may affect the assignment.

Scheduling Methods To describe scheduling methods for loop iterations, we represent each iteration as a node of a directed graph, which will be formally defined as the *iteration dependence graph* (IDG) in Section 5.3.2. Any loop-carried dependence from iteration I to iteration J is represented by a directed edge from node I to node J. An IDG is always acyclic because the directed arcs represent loop-carried dependences, and, for any two dependent iterations, one must be lexically prior to the other.

Wavefront Method: Since any IDG is acyclic, we can define the notion of the *wavefront number* of a node: the maximum path length leading into the node from some source node, where a source node is one that does not have any incoming edge. A *wavefront* is just the set of all nodes with the same wavefront number. Clearly, those iterations of the same wavefront can execute in parallel. Thus if the set of wavefronts of a loop is known either at compile-time or at run-time before entering the loop, it can be used to determine the execution sequence of the iterations beforehand. One technical detail is that a global synchronization between the processors is required to control the progress from one wavefront to the next.

Data-driven Method: Another way of scheduling the iterations is *data-driven*, which allows each iteration to start execution as soon as all its required data become available. In this case, the synchronization among the processors is done locally in a distributed fashion.

The choice of the two methods depends on the nature of the computation and the tradeoffs in the implementation cost. The wavefront method is more suitable for programs with regular data structures, where the schedule can be obtained at compile-time. The advantage is that there will be no overhead to support data-driven execution. For programs with irregular data structures, where the computation load of different iterations in the same wavefront can vary a great deal, the data-driven method usually works better because the wavefront method will incur unnecessary delay due to the global synchronization.

Characterization Using these two orthogonal factors, we can now characterize various compile-time and run-time loop-parallelization techniques. For each of the entries on compile-time techniques in Table 5.1, an example is given. The run-time techniques are described in greater detail since they relate closely to our work.

CW Performing loop skewing followed by iteration space tiling is an example of compiler-time assignment and wavefront scheduling.

CD Assigns iterations to processors randomly. Data-driven scheduling is the natural choice to combine with random assignment.

	Compile-time	run-time Invariant	run-time Dynamic
Wavefront	CW	IW	DW
Data-driven	CD	ID	DD

Table 5.1: Dynamic loop-parallelization techniques characterized by assignment strategies and scheduling methods.

IW Saltz, Mirchandaney, *et al.* [82,97,98] describe a technique where an *inspector* collects data references of a code block or a `DoConsider` loop at run-time, constructs a DAG representing the dependences, and assigns work to processors by partitioning the DAG. An *executor* then computes the code block or the loop on either shared-memory or distributed-memory machines.

ID Johnson, Zukowski and Shea [48] at IBM have developed a parallel circuit simulator running on the Victor multiprocessor systems consisting of an array of transputers. A hand-coded run-time module reads input data structures, analyzes dependences, partitions the loop iteration space and the data structures associated with it, and assigns each portion to a processor. The processors run in a data-driven fashion, sending required circuit information between one another. The module is written with the knowledge of the specific application program as well as properties of input streams.

DW Computing wavefront at the last minute when tasks are dynamically assigned seems hardly worthwhile; using data-driven execution will be much easier and more efficient.

DD Fang, Tang, Yew and Zhu [33] describe a method in which each iteration of the loop is a *task*. It uses data-driven scheduling in the sense that all active tasks, defined to be those whose predecessors have been completed, are placed in a *task pool*. Whenever a processor becomes free, it goes to the task pool and picks up an executable task. The *guided self-scheduling* method from Polychronopoulos and Kuck [87] and the *shortest-delay self-scheduling* method from Tang, Yew and Zhu [104] also fall into this class.

In addition to the above, Zhu and Yew [116] and Midkiff and Padua [80] have developed techniques to insert the synchronization primitives into loops with loop-carried dependence to ensure proper parallel execution. These methods should be combinable with all three assignment strategies.

Our Work The dynamic scheduling approach we are going to describe falls into the class of run-time invariant assignment using either wavefront or data-driven schedule, each requiring different code for house-keeping. We choose to investigate run-time invariant assignment because many important applications such as circuit simulators [101], computational fluid dynamic computation and sparse matrix solvers [27,82,95,97,98,115,114] can be dealt with effectively with this strategy.

5.2.2 Related Work

Comparison with Shared-Memory Model Experimental results [33,87,104] show that the overhead of systems using run-time dynamic assignment with asynchronous data-driven schedule are low on shared-memory machines. But techniques such as the task pool can incur high communication overhead on distributed-memory machines.

Comparison with Saltz *et al.*'s Approach The work by Saltz *et al.* on run-time systems [27,82,95,97,98,115,114] for both shared-memory and distributed-memory machines can be characterized as a pure run-time approach in the sense that all mechanisms (DAG encoding routine, inspector routines, etc.) pertaining to the scheduler are hand-written run-time library routines. Code blocks and variables of the source program are annotated for the needs of run-time support. Their work focuses on scientific applications written in FORTRAN with simple loop structures and reference patterns where the data size rather than redundant references is the main problem.

In contrast, our scheduler is generated by the scheduler generator for each application program, customized by the program slices extracted from the original program. In this scheduler generation framework, we can deal with nested loops where each level can be recursively scheduled over the processors, useful for problems operating over a hierarchy of more and more refined data structures. In addition, compile-time analysis such as redundant reference elimination and dependence analysis can be used to lower the run-time overhead significantly.

We use the following two examples to show the main difference between our and Saltz *et al.*'s work.

Loop Nest 5.1

```
while(topA != NULL) {
   if(...) {
      bar = topA->wB;
      foo = topA->rB;
      bar->rwC->valC = foo->rwC->valC+1;
      bar->valB = foo->valB+2;
      topA->valA = topA->valA+3; }
   else {...}
topA = topA->next; }
```

Loop Nest 5.2

```
doconsider $(i = 1, n)$ {
  do $(j = low(i), high(i))$ {
    $y(i) = y(i) - a(j) * y(column(j))$ } }
```

Our scheduling method is applicable to both Loop Nest 5.1 and Loop Nest 5.2, while Saltz *et al.*'s method is only applicable to the latter. This is because our system includes compilation analysis such as program slicing, redundant redundant reference elimination and dependence analysis of pointers. For Loop Nest 5.1, the effects of these compiler techniques are:

- Dependence analysis of pointers finds that lines 5 and 6, but not the other lines, can cause loop-carried dependence. Therefore, references occurred in lines 5 and 6 need be recorded by the run-time scheduler to find a parallel schedule.
- Redundant reference elimination detects that references in line 5 is *subsumed* (to de defined in Section 5.4) by references in line 6. Therefore, only references occurred in line 6, but not line 5, need be recorded.
- Program slicing finds that lines 1,2,3,4,8 and 9, but not lines 5,6 and 7, can affect reference locations in line 6. (The program slice includes control statements while and IF-THEN-ELSE.) Therefore, lines 1,2,3,4,8 and 9 should be copied into the run-time scheduler to compute the reference locations to be recorded.

Clearly, these compiler techniques are necessary for sending the run-time scheduler for Loop Nest 5.1. In addition, dependence analysis and redundant reference elimination can reduce the number of references to be recorded, and, therefore, reduce the run-time scheduling overhead for applications with many redundant references and references that do not cause loop-carried dependences. Two examples of applications in this category are the circuit simulator and the fluid dynamics kernel presented in Section 5.5. The scheduling overhead of the circuit simulator is 0.63% of the sequential execution time of the source code. And the overhead is 6.1% for the fluid dynamics kernel.

Experimental results from Saltz *et al.* [98] show that the overhead of their run-time scheduler ranges from 20% to 60% for applications like the one shown in Loop Nest 5.2. For such applications which do not have redundant references and complex control structures in the first place, our compiler techniques would not really help. So the compiler techniques in our system are not in Saltz *et al.*'s system. However, for large programs with complex control structures and many redundant references like the circuit simulator, compiler techniques are crucial for this run-time approach to be workable in practice.

Comparison with Work on Redundant Dependence Elimination Li and Abu-Sufah [73], Midkiff and Padua [80], and Krothapalli and Sadayappan [57] have studied *redundant dependence elimination* techniques in the context of parallelizing loops by inserting synchronization primitives. Such analysis reduces the number of synchronizations issued.

Our work is motivated by distributed-memory machines where communication costs must be carefully considered. We consider iteration-level parallelism, which is more coarse-grained than the shared-memory approach. Since data must be partitioned, it is more important to collect dependent iterations into the same processor to minimize communication. The focus will be on distributing the independent iterations over different processors as opposed to overlapping the execution of dependent iterations. These differences in perspective result in quite different techniques for reducing redundant references. We will discuss these points further.

Comparison with Work on Access Anomaly Detection Dinning and Schonberg [31] have studied methods for monitoring parallel program execution to detect *access anomalies.* An access anomaly occurs when either two concurrent threads both write or one reads and one writes a shared memory location without coordinating these accesses. Because they do not use static compile-time analysis to reduce redundant references, and because the monitoring overhead is compared with the timing of parallel program execution, their experimental results show that monitoring entails a 3-fold to 6-fold slowdown.

In comparison, the iteration-level dependence used in our method is much more coarse-grained than access anomaly detection. Furthermore, our experiments show that redundant reference elimination and dependence analysis do greatly reduce the run-time scheduling overhead for restricted nonaffine loops (defined in Section 5.1.1).

A similarity between their and our methods is that we use the same technique to save memory in storing reference histories. This technique will be discussed with the *read set* and *last write* in Section 5.3.3.

5.3 Scheduler Generator

5.3.1 System Overview

The hybrid system consists of a run-time component and a compile-time component as shown in Figure 5.1. The run-time component consists of, in addition to the target parallel program, an *IDG-constructor*, an *IDG-partitioner*, and a *data partitioner* customized for the source program.

The IDG-constructor builds the iteration dependence graph (IDG) at run time. The IDG-partitioner then assigns and schedules loop iterations by performing graph-partitioning on the IDG. By the inverse of the so-called owner-compute rule of SPMD style programs, the input data structure is partitioned according to the iteration partition over the IDG. By "inverse" we mean that the processor which computes the iteration gets assigned a copy of the corresponding data. Together, these three parts are referred to as the *run-time scheduler.*

The compiler consists of two main components: a *schedule generator* and a *parallel-program generator.* We do not describe the parallel-program generator in this chapter, which is an extension of the work by Li and Chen [67] and closely relates to the work by Saltz *et al.* [27,82,95,97,98,115,114] The focus is on the scheduler generator, and in particular, the one with low run-time overhead. To construct the IDG, read and write references of the source program must be recorded at run-time. If one does this naively, the space and time may both blow up for large programs. The idea behind our approach is to examine the minimum number of references necessary to construct an IDG. Some references such as pointer dereferencing and induction variables do not create dependence. Others form equivalence classes whereby only representative ones

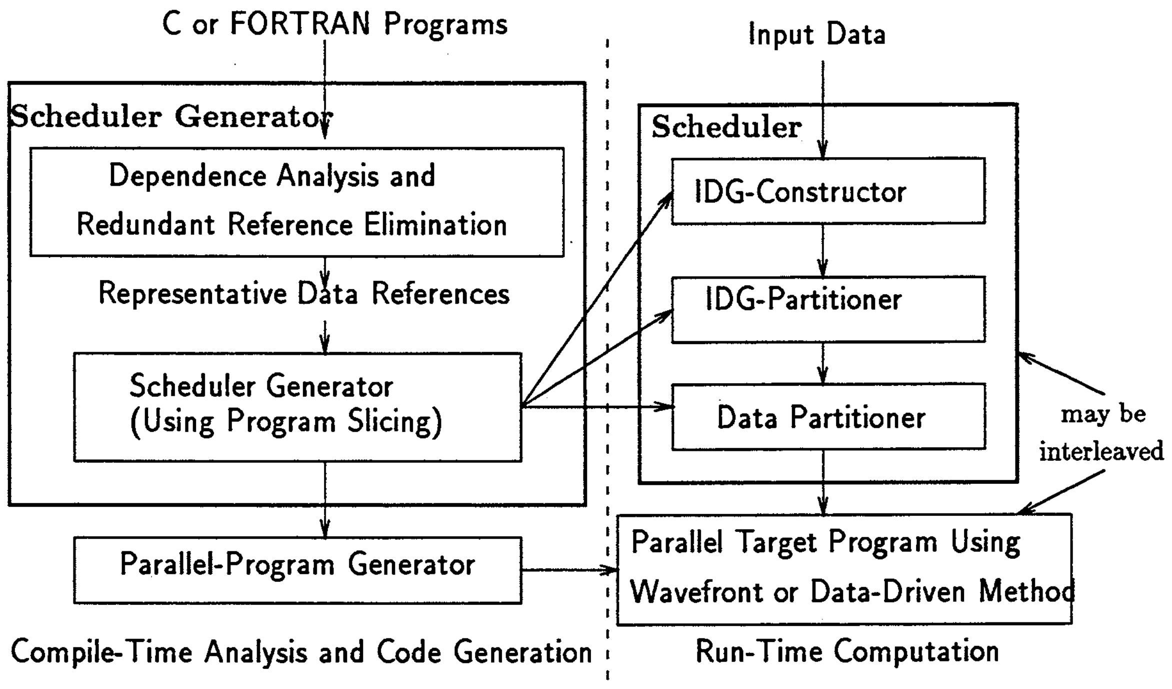
C or FORTRAN Programs
Scheduler Generator
Dependence Analysis and Redundant Reference Elimination
Representative Data References
Scheduler Generator (Using Program Slicing)
Parallel-Program Generator
Compile-Time Analysis and Code Generation
Input Data
Scheduler
IDG-Constructor
IDG-Partitioner
Data Partitioner
may be interleaved
Parallel Target Program Using Wavefront or Data-Driven Method
Run-Time Computation

Figure 5.1: The structure of the hybrid compiler/run-time parallelization system.

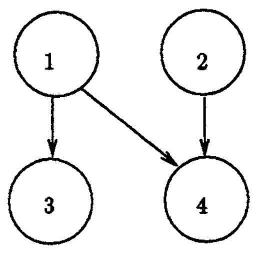

Figure 5.2: An example of the iteration dependence graph.

need to be recorded.

In the following, we first describe the IDG representation, then the primitives for scheduler generation, and finally the methods and optimizations. A C programming example is also given.

5.3.2 Iteration Dependence Graph (IDG)

We define the *iteration dependence graph* (IDG for short) of a loop L to be a directed graph with one vertex for each iteration of L. There is an edge from node J to node K, denoted by $J \Rightarrow K$, in the IDG if and only if for some statements S_1 and S_2 in the loop a loop-carried dependence $S_1@J \Rightarrow S_2@K$ exists in L. (A dependence $S_1@J \Rightarrow S_2@K$ is loop-carried if $J \prec K$ as defined in Section 2.2.) Since only loop-carried dependences are presented in the IDG, $J \prec K$ must hold for an edge $J \Rightarrow K$ in the IDG. Therefore, any IDG is acyclic.

Recall that "$\overset{*}{\Rightarrow}$" is the reflexive and transitive closure of the dependence relation "$\Rightarrow$" over iterations as defined in Chapter 2. If $J \overset{*}{\Rightarrow} K$ holds, then iterations J and K are dependent. If neither $J \overset{*}{\Rightarrow} K$ nor $K \overset{*}{\Rightarrow} J$ holds, then iterations J and K are independent.

An example of an IDG is shown in Figure 5.2. Iterations are assigned to processors by partitioning the IDG. In order to reduce communication overhead for distributed-memory machines, dependent iterations, e.g. iterations 1 and 3, should be aggregated to the same processor. Additionally, in order to maximize parallelism, independent iterations, e.g. iterations 1 and 2, should be distributed to different processors.

This representation is similar to the DAG representation in [17,82,96], except that an IDG is defined with respect to a loop nest while the others are defined with respect to a set of recursive definitions or an annotated code block.

We now discuss how to generate the most important part of the run-time scheduler: the IDG-constructor.

5.3.3 IDG-Constructor

As discussed above, the IDG is built at run-time by the IDG-constructor, which is generated by the compiler. The IDG-constructor records read and write references to build the IDG. We now define the notion of *read set* and *last write* to capture the relationship between data references and loop iterations.

Read Set and Last Write Let X be a *memory location* being referenced. We define $\mathcal{R}(X)$, called the *read set* for X, to be a set containing the iterations I in the iteration space at which X is read after X is last written. If X is read more than once at iteration I, then only one instance is in the set $\mathcal{R}(X)$.

Similarly, we define $\mathcal{W}(X)$, called the *last write* for X, to be the iteration at which X is last written, or null if X is not yet written.

Note that, for FORTRAN, "memory location" is replaced by "array name" applied to an "array index".

Recording Procedure For each reference in the source program, the compiler generates a call to a procedure record_dep which updates the read set and the last write, deduces dependences and constructs edges in the IDG. Procedure record_dep has three parameters: the first two parameters X and I indicate that memory location X is referenced at iteration I, and the third parameter is either $'r'$ for a read reference or a $'w'$ for a write reference.

A call to procedure record_dep$(X, I, 'r')$ is for recording possible flow dependence caused by a read reference to location X at iteration I. Clearly, I should be added to the read set $\mathcal{R}(X)$. If the last write $\mathcal{W}(X)$ is not null, i.e. $J = \mathcal{W}(X)$, then location X is last written at iteration J and the flow dependence $J \Rightarrow I$ exists, which implies an edge in the IDG. On the other hand, if $\mathcal{W}(X)$ is null, then X has not yet been written and there is no flow dependence.

A call to procedure record_dep$(X, I, 'w')$ is for recording possible output and anti-dependences caused by a write reference to location X at iteration I. If the last write $\mathcal{W}(X)$ is not null, i.e. $J = \mathcal{W}(X)$, then X is also written at iteration J and the output dependence $J \Rightarrow I$ exists. Now I is the last iteration which writes X; therefore, the last write $\mathcal{W}(X)$ should be assigned a new value I. On the other hand, if the last write $\mathcal{W}(X)$ is null, then there is no output dependence and we only need to assign I to $\mathcal{W}(X)$.

Similar discussion holds for anti-dependences as follows. If the read set $\mathcal{R}(X)$ is not empty, then for all J in $\mathcal{R}(X)$, the anti-dependence $J \Rightarrow I$ exists. In addition, the read set $\mathcal{R}(X)$ should be reset to the empty set because there is no new read after the latest write at iteration I. To summarize, procedure record_dep is defined in Figure 5.3.

An Example for Dependences Consider the following example: Let a location X be read at iterations 1, 3 and 4, and be written at iterations 2 and 5 as shown in Figure 5.4. The results from calling record_dep at these iterations are:
After iteration 1, no dependence is found, $\mathcal{R}(X) = \{1\}$ and $\mathcal{W}(X) = \textit{null}$.
After iteration 2, anti-dependence $1 \Rightarrow 2$ is found, $\mathcal{R}(X) = \{\}$ and $\mathcal{W}(X) = 2$.
After iteration 3, flow dependence $2 \Rightarrow 3$ is found, $\mathcal{R}(X) = \{3\}$ and $\mathcal{W}(X) = 2$.
After iteration 4, flow dependence $2 \Rightarrow 4$ is found, $\mathcal{R}(X) = \{3, 4\}$ and $\mathcal{W}(X) = 2$.
After iteration 5, output dependence $2 \Rightarrow 5$ and anti-dependences $3 \Rightarrow 5$ and $4 \Rightarrow 5$ are found, $\mathcal{R}(X) = \{\}$ and $\mathcal{W}(X) = 4$.

We now use an example to illustrate how to generate the IDG-constructor with

```
record_dep $(X, I, c)$
$X$:a reference location
$I$: an iteration
$c$: $'r'$ or $'w'$
  if $(c =' r')$
            $\mathcal{R}(X) = \mathcal{R}(X) \cup \{I\}$
            if $(\mathcal{W}(X) \neq null)$
              then $J = \mathcal{W}(X)$
                   there is a flow dependence $J \Rightarrow I$
                   (an edge from $J$ to $I$ in the IDG)
  if $(c =' w')$
            if $(\mathcal{W}(X) \neq null)$
              then $J = \mathcal{W}(X)$
                   the output dependence $J \Rightarrow I$ exists
            $\mathcal{W}(X) = I$
            for all $J \in \mathcal{R}(X)$
              the anti-dependence $J \Rightarrow I$ exists
            $\mathcal{R}(X) = \{\}$
```

Figure 5.3: The recording procedure.

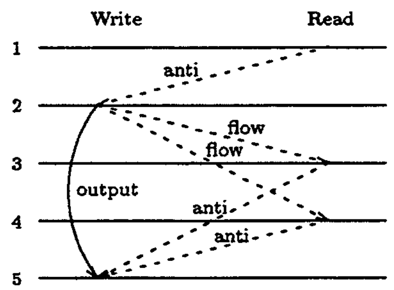

Figure 5.4: An example of dependences.

calls to these two procedures.

An Example for IDG-constructor The example C program with pointers is in Figure 5.5. We want to parallelize the while loop from line 11 to line 17.

Functionality of the Program The relationships among structures and pointers are shown in Figure 5.6. Instances of structure `A` form a linked list, and similarly for instances of structure `B`. Each instance of `A` has two pointers `rB` and `wB` pointing to some instances of `B`, which can be different. (We use `r` for labeling a read access, `w` for labeling a write access, and `wr` for labeling an access which can be either read or write.) These two pointers take on values returned by the function `lookup`, which finds a particular instance of `B` that matches the identifier given as the first argument of `lookup` as shown in lines 8 and 9 of the code. The identifiers `id1` and `id2` are read from the input stream in line 7. Each instance of `B` has a pointer `rwC` pointing to an instance of structure `C` created by the `malloc` function.

The `while` loop goes through every instance of structure `A` where the current one is pointed by `topA`. Lines 14, 15 and 16 in the loop body read three integers to compute three other integers contained in the instances of `A`, `B` and `C` respectively.

Note that, since the pointers from instances of `A` to instances of `B` are input dependent, and dependences are determined by those pointers, this while loop cannot be parallelized by static compile-time analysis.

Algorithm for Generating A Naive IDG-Constructor We now focus on the while loop (line 11-17) which is repeated below for easy reference.

Loop Nest 5.3

The Source Loop from Figure 5.5:

```
11  while(topA != NULL) {
12       bar = topA->wB;
13       foo = topA->rB;
```

```
typedef struct C {
    int valC;
    other fields ...} *pointerC;
typedef struct B {
    int idB, valB;
    pointerC rwC;
    struct B *next; } *pointerB;
typedef struct A {
    int valA;
    pointerB rB, wB;
    struct A *next; } *pointerA;
main()
{    int id1, id2, nA, nB, i;
     pointerA nodeA, topA;
     pointerB nodeB, topB, foo, bar;
     pointerC nodeC;

     scanf("%d %d",&nB,&nA);
  0  for(i=0; i<nB; i++) {   /* initialize B and C */
  1     nodeB = (pointerB) malloc(sizeof *nodeB);
  2     nodeC = (pointerC) malloc(sizeof *nodeC);
  3     scanf("%d %d %d",&(nodeB->idB),&(nodeB->valB),&(nodeC->valC));
  4     nodeB->rwC = nodeC; nodeB->next = topB; topB = nodeB;}
  5  for(i=0; i<nA; i++) {   /* initialize A */
  6     nodeA = (pointerA) malloc(sizeof *nodeA);
  7     scanf("%d %d %d",&(nodeA->valA),,&id1,&id2);
  8     nodeA->rB = (pointerB) lookup(id1,topB); /* find B instances */
  9     nodeA->wB = (pointerB) lookup(id2,topB); /* matching the id */
 10     nodeA->next = topA; topA = nodeA; } /* build list */

 11  while(topA != NULL) {
 12     bar = topA->wB;
 13     foo = topA->rB;
 14     bar->rwC->valC = foo->rwC->valC+1;
 15     bar->valB = foo->valB+2;
 16     topA->valA = topA->valA+3;
 17     topA = topA->next; } }

int lookup(id,topB)
int id; pointerB topB;
{     pointerB nodeB;
      nodeB = topB;
      while(nodeB != NULL) {
 if(nodeB->idB == id) return(nodeB);
 nodeB = nodeB->next; }}
```

Figure 5.5: An example program with pointers.

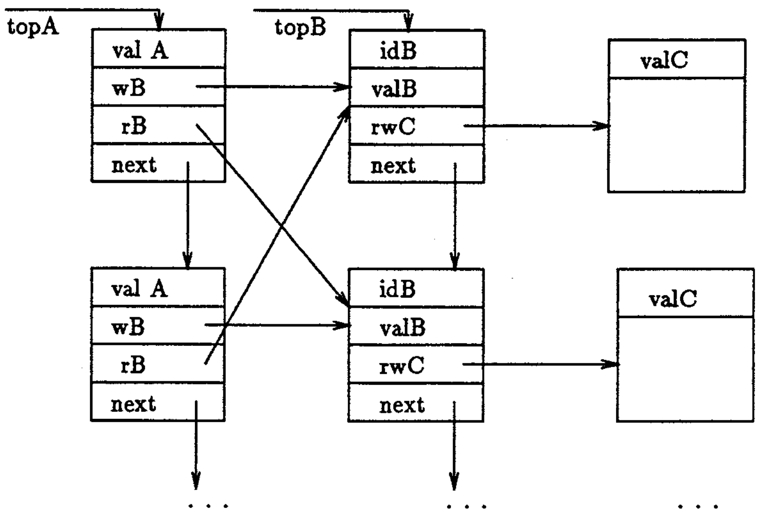

Figure 5.6: Relationship among structures and pointers of the example.

```
14      bar->rwC->valC = foo->rwC->valC+1;
15      bar->valB = foo->valB+2;
16      topA->valA = topA->valA+3;
17      topA = topA->next; }
```

Nodes of the IDG For each instance of the iteration space of the loop, a call to procedure **create_node** is generated which creates a node corresponding to the current iteration instance in the IDG. For identifying these nodes, we need an explicit loop iteration counter. In this example, variable i is used as the iteration counter. Clearly, i should be incremented for each iteration, and i is used as a parameter for the procedure calls **create_node** and **record_dep**.

Edges of the IDG For each reference, a call to procedure **record_dep** is generated which creates edges between the nodes of the IDG. For example, a read reference to location $\&(topA \rightarrow wB)$ in line 12 implies a call to **record_dep**$(\&(topA \rightarrow wB), i, 'r')$.

Program Slices In addition to generating those procedure calls, we also need to copy some statements from the source loop to the IDG-constructor as follows. If a variable is necessary to determine the reference location, e.g. *bar* is used in the call to record_dep($\&(bar \rightarrow rwC \rightarrow valC), i, 'w'$), then the set of statements that directly or indirectly contributed to the value of that variable needs to be copied from the source program to the IDG-constructor. Similarly for control statements like **while** and **if-then-else**, since they determine whether a location is referenced or not.

The technique of *static program slicing* [105], which is based on data flow analysis and execution trace, can be used to find the program slice with respect to one variable. A program slice with respect to a set of variables can be obtained by taking the union of slices with respect to individual variables in the set. In this example, the program slice we need consists of statements in lines 12, 13 and 17 of the source program which compute *bar*, *foo* and *topA*. The resulting naive IDG-constructor is shown in Loop Nest 5.4.

Loop Nest 5.4

A Naive IDG-Constructor:

```
  i = 0;
  while (topA ≠ NULL){                                    /* 11 slicing */
      i++; /* explicit iteration counter */
      create_node(i); /* create a node of the IDG */
      bar = topA → wB;                                    /* 12 slicing */
      foo = topA → rB;                                    /* 13 slicing */
      record_dep(&(topA → wB), i,'r');
      record_dep(&bar, i,'w');                            /* 12 recording */
      record_dep(&(topA → rB), i,'r');
      record_dep(&foo, i,'w');                            /* 13 recording */
      record_dep(&(foo → rwC → valC), i,'r');
      record_dep(&(bar → rwC → valC), i,'w');             /* 14 recording */
      record_dep(&(foo → valB), i,'r');
      record_dep(&(bar → valB), i,'w');                   /* 15 recording */
      record_dep(&(topA → valA), i,'r');
      record_dep(&(topA → valA), i,'w');                  /* 16 recording */
      record_dep(&(topA → next), i,'r');
      record_dep(&topA, i,'w');                           /* 17 recording */
      topA = topA → next;                                 /* 17 slicing */}
```

For program slicing including control statement, let us look at the following example. The source code is on the left-hand side, and the corresponding IDG-constructor is on the right-hand side, which includes both conditional expressions as shown below:

Loop Nest 5.5

```
while (...){
    if($a > 0$)  $b \rightarrow c = \ldots$
    else  $b \rightarrow d = \ldots$        }
```

Loop Nest 5.6

```
$i = 0$;
while (...){
    $i++$;
    if($a > 0$)  record_dep(&($b \rightarrow c$), $i$, 'w')
    else  record_dep(&($b \rightarrow d$), $i$, 'w')
    ...                                      }
```

Another point to mention is that we need *static* program slicing [105] instead of *dynamic* program slicing [1,56]. The difference between the two is that the static slice is the set of all statements that might affect the value of a given variable for *any* program input, while the dynamic slice consists of all statements that actually affect the value of a variable for a *given* program input. We need static slicing because the scheduler should work for any program input. For loops with procedure calls, we need *interprocedural program slicing* [47].

5.3.4 Optimizing IDG-Constructor

Dependence Analysis and RRE The naive IDG-constructor constructs the IDG by recording all data references. In fact, some references do not contribute to edges in the IDG and some other references contribute to redundant edges in the IDG. All those references are called *redundant references* and should not be made into the IDG-constructor.

For this example, dependence analysis on pointers [16,42,44,46,49,62,63] tells us that

- Instances of structure A do not cause loop-carried dependence.

- *foo* and *bar* are used for pointer dereferencing; they do not cause true dependences.
- *topA* is an "induction" variable for navigating through the linked list.

In addition, RRE says that instances of structures B and C cause identical dependences, and therefore, only the references to instances of either `B` or `C` need to be recorded. After removing those recording procedure calls for redundant references, the resulting IDG-constructor is as shown in Loop Nest 5.7. This IDG-constructor is much smaller and uses much less space for the read sets and the last writes.

Loop Nest 5.7

An IDG-constructor with redundant references being removed:

```
i = 0;
while (topA ≠ NULL){                        /* 11 slicing */
      i++;   create_node(i);
      bar = topA → wB;                      /* 12 slicing */
      foo = topA → rB;                      /* 13 slicing */
      record_dep(&(foo → valB), i, 'r');
      record_dep(&(bar → valB), i, 'w');    /* 15 recording */
      topA = topA → next;                   /* 17 slicing */}
```

Statement Substitution Note that, *foo* and *bar* in the source program shown in Figure 5.5 are read twice in lines 14 and 15. Therefore, using these two variables for pointer dereferencing can avoid recomputing $topA \rightarrow wB$ and $topA \rightarrow rB$. However, after removing redundant references, *foo* and *bar* are only read once in the IDG-constructor as shown in Loop Nest 5.7. In this case, using them for pointer dereferencing in unnecessary and they can cause extra `load` and `store` instructions in the compiled object code of the IDG-constructor. This situation can be avoided by *statement substitution* [110] as follows. Each read reference to *foo* is substituted by $topA \rightarrow wB$ and each read reference to bar is substituted by $topA \rightarrow rB$, and the two statements computing foo and bar are removed. After statement substitution, the optimized IDG-constructor is shown in Loop Nest 5.8.

Loop Nest 5.8

An optimized IDG-constructor:

```
$i = 0;$
while ($topA \neq NULL$){                              /* 11 slicing */
    $i++;$   create_node($i$);
    record_dep(&($topA \rightarrow rB \rightarrow valB$), $i$,'r');
    record_dep(&($topA \rightarrow wB \rightarrow valB$), $i$,'w');  /* 15 recording */
    $topA = topA \rightarrow next;$                      /* 17 slicing */}
```

5.3.5 IDG-Partitioner and Data Partitioner

Once the IDG is constructed, an IDG-partitioner partitions the nodes of the IDG into disjoint sets such that dependent iterations are aggregated into the same processor as much as possible and independent iterations are distributed to different processors. This is as hard as the *minimum cut* problem which is NP-complete [39]. Therefore, we use a heuristic to do the partition. How to choose a good heuristic algorithm is beyond the scope of this thesis.

Partition on IDG induces a partition on the data using the inverse of the so-called owner-compute rule: $A(X(I))$ will be written by processor P which is assigned iteration I.

5.3.6 Summary of Scheduler Generating Steps

To summarize, the compiler generates the run-time scheduler in the following steps:

1. Use dependence analysis and redundant reference elimination techniques to obtain representative references in the source loop.

2. For each representative reference, generate a call to the recording procedure **record_dep** in the IDG-constructor.
3. Find the program slice which computes the values of the variables necessary to determine the reference locations.
4. Generate a code based on a chosen heuristic for partitioning the IDG into p disjoint subsets, where p is the number of processors of the target distributed-memory machine.
5. Generate a code for data partitioning using the inverse of the owner-compute rule.

5.4 Redundant Reference Elimination (RRE)

We now discuss how to eliminate redundant references to reduce the run-time overhead of the scheduler. We consider both array and pointer references.

5.4.1 Common References

Common References to Arrays and Fields If an array is read more than once in the loop body with common subscript expressions, then we say these reads are *common references.* Similarly for multiple writes. Clearly, all except one common reference is redundant. Therefore, the IDG-constructor only needs to record one reference from a set of common references.

Similarly, if a field of an instance of a structure is read (written) several times in the loop body, all except one reference is redundant.

Common References to Multiple Arrays and Fields Common references to a single array or a single field can be generalized to multiple arrays and multiple fields of the same instance of a structure due to the underlying assumptions that (1) if arrays have common references, then they will be aligned and assigned to processors in the same way, and (2) an entire instance of a structure will be assigned to the same processor. For example, arrays A and B and fields x and y have common references in the following loops:

Loop Nest 5.9

```
DO (I:E){
  A(X(I)) = ...
  ... = A(Y(I))
  B(X(I)) = ...
  ... = B(Y(I)) }
```

Loop Nest 5.10

```
while (...){
  a = ...
  a → x = ...
  a → y = ... }
```

Common References under Simultaneous Permutation of Components

Theorem 5.1 (Simultaneous permutation of components)
Common references under simultaneous permutation of components are redundant.

Consider the following loop:

Loop Nest 5.11

```
DO (I:E){
  A(X(I),Y(I)) = ...
  ... = A(U(I),V(I))
  B(Y(I),X(I)) = ...
  ... = B(V(I),U(I)) }
```

This theorem says that the references to either A or B, but not both, are redundant.

5.4.2 Subsumed References

Partial Reference If some components of subscripts are ignored in dependence analysis, then we call it *partial dependence analysis.* An array reference with some components of subscripts being ignored is called a *partial reference.* The components used in partial reference and partial dependence analysis are called *critical components.*

It is easy to see that partial dependence analysis is always conservative [113]. This is because considering more components will further differentiate the reference locations and make dependence less possible. Consider the following loops:

Loop Nest 5.12

```
DO (i = 1,n){
  A(i,B(i)) = ...
  ... = A(i,C(i)) }
```

Loop Nest 5.13

```
DO (i = 1,n){
  A(i) = ...
  ... = A(i) }
```

Let the first component of the subscripts of array A be the critical component. After ignoring the second component, Loop Nest 5.12 becomes Loop Nest 5.13 above. No loop-carried dependence in Loop Nest 5.13 implies no loop-carried dependence in Loop Nest 5.12, and loop-independent dependence in Loop Nest 5.13 implies that loop-independent dependence may exist in Loop Nest 5.12. In fact, there is loop-independent dependence in Loop Nest 5.12 only when i exists in $[1,n]$ such that $B(i) = C(i)$.

Subsumed Partial Reference

Theorem 5.2 (Subsumed partial reference)
If the partial references at critical components to array A are subsumed by complete references of the same symbolic forms to array B, then references to array A are redundant.

This is because references to B will cause more conservative dependences than references to A. Consider the following loop:

Loop Nest 5.14

```
DO $(I{:}E)$ {
   $B(X(I)) = \ldots$
   $\ldots = B(Y(I))$
   $A(X(I), U(I)) = \ldots$
   $\ldots = A(Y(I), V(I))$ }
```

By choosing the first component as the critical component of subscripts of array A, Theorem 5.2 says that references to A are redundant.

Subsumed Pointer Reference To obtain a pointer version of Theorem 5.2, we need to review some definitions.

A *structure* is an object composed of a collection of *fields*. A collection of structures can be modeled by a directed graph $G = (V, E)$, which is called a *structure graph* in [62,63]. Each vertex a in V corresponds to an instance of a structure. We use a to denote both the vertex and the structure instance represented by a. Each edge in E from vertex a to vertex b indicates that the structure instance a contains a pointer in a field to the structure instance b.

A directed graph is *rooted* at vertex r if there is a path from r to every vertex in the graph [2]. We assume that the structure graph is rooted. (Otherwise, we just add a root and some edges to make it rooted.) Vertex a is a *dominator* of vertex b if every path from the root to b contains a [2].

Theorem 5.3 (Pointer version of Theorem 5.2)

Let vertex a be a dominator of vertex b in the structure graph. Let x be a field of

a. If the fields of b are read only when x is read and the fields of b are written only when x is written, then the references to all fields of b are redundant.

In the following example, if a is a dominator of b in the structure graph, then Theorem 5.3 says that write references to field y of b is redundant:

Loop Nest 5.15

```
while (...){
   a = ...
   a → x = ...
   a → b → y = ... }
```

This theorem can be used to detect that references to `foo→rwC→valC` and `bar→rwC→valC` are redundant in the code in Figure 5.5.

5.5 Applications of the Dynamic Scheduler Generator

We now demonstrate the effectiveness of our hybrid compiler/run-time scheduling method by applying it to two realistic application programs. We first define a specific class of loops, *the restricted nonaffine loops*, for which the hybrid method is more suitable because the run-time scheduling overhead for such loops can be reduced to an insignificant level.

5.5.1 Restricted Nonaffine Loops

We define *restricted nonaffine loops* to be a class of loops consisting of *iterative loops*, and loops with *loop-invariant pointers* and much smaller IDG-constructors compared to the loops themselves. Clearly, a much smaller IDG-constructor requires much less computation time than the source loop. We now discuss loop-invariant pointers and iterative loops.

Loop-Invariant Pointers If pointers and indirect array reference locations are computed before the computation of the loop nest and they do not change during the computation of the loop nest, then we call them *loop-invariant* pointers. Otherwise, they are *loop-variant* pointers.

Our method is more suitable for loop nests with loop-invariant pointers because the scheduler only needs to record, but not compute, the reference locations. For loop nests with loop-variant pointers, recording and computing reference locations may result in large scheduling overhead. Consider the following loop nest:

Loop Nest 5.16

```
DO $(j = 1, n)$ {
    $A(B(j)) = \ldots$
    $B(A(j)) = \ldots$ }
```

Since the two statements in the loop body are in the same dependent block and reference locations are determined by the computation of these two statements, the program slice must contain both statements to compute the reference locations. With the program slice and calls to recording procedures, the scheduler of Loop Nest 5.16 is shown in Loop Nest 5.17 below, where "array name" and "array index" are used as parameters to the recording procedure record_dep. Clearly, the scheduling overhead of this example is higher than computing the loop itself.

Loop Nest 5.17

```
DO ($j = 1, n$) {
    record_dep("$B$", $j$, $j$, 'r')   record_dep("$A$", $B(j)$, $j$, 'w')
    $A(B(j)) = \ldots$
    record_dep("$A$", $j$, $j$, 'r')   record_dep("$B$", $A(j)$, $j$, 'w')
    $B(A(j)) = \ldots$                                  }
```

Iterative Loop Nests Consider the following loop nest:
Loop Nest 5.18

```
DO ($i = 1, iter$) {
    Loop Nest $\mathcal{L}$ }
```

If Loop Nest $\mathcal{L}$ only contains loop-invariant pointers, then we call Loop Nest 5.18 an *iterative loop nest.* That is, data reference locations, and therefore data dependences, in Loop Nest $\mathcal{L}$ are invariant to different iterations of the outer loop. In this case, we only need to schedule Loop Nest $\mathcal{L}$ once and the same schedule can be used *iter* times for the parallel execution of Loop Nest $\mathcal{L}$. Therefore, the overhead of the scheduler is amortized among those iterations. The scheduler and parallel code of Loop Nest 5.18 is:
Loop Nest 5.19

```
scheduler of Loop Nest $\mathcal{L}$
DO ($i = 1, iter$) {
    parallel code of Loop Nest $\mathcal{L}$ }
```

5.5.2 Optimizing Data Layout

Our hybrid method is effective not only for detecting parallelism but also for optimizing data layout to reduce communication overhead. This effect is illustrated by the fluid dynamics kernel presented in Section 5.5.4 below. For this example, we compare the communication time associated with two data layouts: one is a regular block partition, and the other is an irregular data layout obtained by the scheduler. We find that the irregular data layout results in less communication time, and the communication time saved is larger than the scheduling overhead. Therefore, the total execution time is reduced when using the scheduler.

Partitioner If the application is readily parallelizable and the main concern is to find a good irregular data layout, then the recording procedures in the schedulers should be modified accordingly. For obtaining a good data layout, it does not matter whether a reference is either a read or a write. That is, the last parameter $'r'$ or $'w'$ of procedure record_dep can be eliminated. In addition, since all iterations are readily parallelizable, and the main purpose for constructing an IDG is to detect parallelism, an IDG is not necessary for finding a good data layout. Therefore, procedure record_dep is modified to assign the current iteration and data elements referenced in the current iteration to processors directly. This modification saves the overhead for constructing the IDG. Since the IDG-constructor is not a suitable name in this case, we call it the *partitioner*. Again, how to choose a good mapping strategy is beyond the scope of this thesis.

We now present the two application programs: a circuit simulator and a fluid dynamics kernel. Both applications are iterative loops with loop-invariant pointers and small IDG-constructors.

5.5.3 Circuit Simulator

The first application is a waveform-relaxation circuit simulator [28] developed at IBM by Shea, Johnson and Zukowski [48,101]. The simulator is a 12,000-line C program with pointers, where the example program in Figure 5.5 captures major pointer dependences in the source code. The main concern is to find parallelism out of loop iterations. The run-time scheduler is generated by hand-compilation according to dependence analysis of pointers, program slicing and RRE, and the parallel circuit simulator is coded by Johnson and Zukowski.

Experimental Result The experimental timing of executing the circuit simulation loop for a specific circuit is collected as follows, after being normalized according to transputer timing:

- The sequential loop takes 18000 seconds on one transputer.
- The run-time scheduler takes 112.95 seconds on one transputer.
- The parallel loop takes 99.45 seconds on Victor, the 256-node transputer array developed at IBM.

The scheduling overhead is 0.63% (112.95/18000) of the sequential execution time, and it is 113.6% (112.95/99.45) of the parallel execution time. Note that the overhead is significantly amplified by parallelization. This is another reason why using compile-time analysis to reduce the run-time overhead is critical for massively parallel machines.

5.5.4 Fluid Dynamics Kernel

Source Code The second application is a fluid dynamics kernel taken from a program that computes convective fluxes using a method based on Roe's approximate Riemann Solver [106,107,27,95,115,114]. We transform the source code from ARF (ARguably FORTRAN) [27,95,115,114] to C and the result is given in Figure 5.7. This loop is readily parallelizable because the only loop carried dependences are caused by lines 36 and 37 with *accumulative* operations. Therefore, the main concern is to generate a partitioner which will find a good data layout at run time to reduce communication overhead.

```
for(i=0;i<medsiz;i++) {
 1  n1=indx[0][i];
 2  n2=indx[1][i];
    for(k=0;k<4;k++) {
 4    q1[k]=qq[k][n1];
 5    q2[k]=qq[k][n2]; }
    rnx=0.1; rny=0.1; rmag=0.1;
    rhol=q1[0]; ul=q1[1]; vl=q1[2]; pl=q1[3];
    rhor=q2[0]; ur=q2[1]; vr=q2[2]; pr=q2[3];
    ubr=rnx*ur+rny*vr; ubl=rnx*ul+rny*vl;
    hr=gamma1*pr/(rhor*gm1)+(ur*ur+vr*vr)*0.5;
    hl=gamma1*pl/(rhol*gm1)+(ul*ul+vl*vl)*0.5; r12=(rhor/rhol);
    rho12=r12*rhol; r121=1./(r12+1.); u12=(r12*ur+ul)*r121;
    v12=(r12*vr+vl)*r121; ub12=(r12*ubr+ubl)*r121; h12=(r12*hr+hl)*r121;
    c12=(gm1*(h12-(u12*u12+v12*v12)*0.5)); drho=rhor-rhol;
    du=ur-ul; dv=vr-vl; dp=pr-pl; dub=ubr-ubl;
    alam1=fabs(ub12); alam2=fabs(ub12+c12); alam3=fabs(ub12-c12);
    cvar1=drho-dp/(c12*c12); qspd2=(u12*u12+v12*v12)*0.5;
    c1221=1./(2.*c12*c12);
    f11=cvar1*alam1;
    f12=(u12*cvar1+rho12*(du-rnx*dub))*alam1;
    f13=(v12*cvar1+rho12*(dv-rny*dub))*alam1;
    f14=(qspd2*cvar1+rho12*(u12*du+v12*dv-ub12*dub))*alam1;
    f21=(dp+rho12*c12*dub)*c1221*alam2;
    f22=f21*(u12+c12*rnx);
    f23=f21*(v12+c12*rny);
    f24=f21*(h12+c12*ub12);
    f31=(dp-rho12*c12*dub)*c1221*alam3;
    f32=f31*(u12-c12*rnx);
    f33=f31*(v12-c12*rny);
    f34=f31*(h12-c12*ub12);
    flx[0]=rmag*(rhor*ubr+rhol*ubl-f11-f21-f31)*0.5;
    flx[1]=rmag*(rhor*ur*ubr+pr*rnx+rhol*ul*ubl+pl*rnx-f12-f22-f32)*0.5;
    flx[2]=rmag*(rhor*vr*ubr+pr*rny+rhol*vl*ubl+pl*rny-f13-f23-f33)*0.5;
    flx[3]=rmag*(hr*ubr*rhor+hl*ubl*rhol-f14-f24-f34)*0.5;
    for(k=0;k<4;k++) {
 36    res[k][n1]=res[k][n1]-flx[k];
 37    res[k][n2]=res[k][n2]+flx[k]; }
}
```

Figure 5.7: Fluid dynamics kernel.

Partitioner We have implemented a partitioner generator which takes C programs with indirect array references as input and performs dependence analysis, program slicing, RRE and statement substitution to generate run-time partitioners.

To generate the partitioner for the fluid dynamics kernel, the compile-time analysis is as follows:

- Dependence analysis tells us that variables $n1$ and $n2$, computed in lines 1 and 2, induce indirect array references in lines 4, 5, 36 and 37, and the program slice affecting indirect array references consists of lines 1 and 2 only.

- RRE detects that references to either qq in lines 4 and 5 or res in lines 36 and 37 are redundant since they are common references. Note that, the first dimension of qq (or res) is referenced with loop index k; therefore, we only need to record indirect references to the second dimension of qq (or res). In addition, since qq (or res) is the only variable to be recorded, parameter "qq" is in fact redundant for procedure record_dep in this special case.

- Since $n1$ and $n2$ are used only once for pointer dereferencing in the partitioner, they should be removed by statement substitution to avoid extra `load` and `store` instructions as discussed before.

To summarize, the partitioner for the fluid dynamics kernel is shown in Loop Nest 5.20.

Loop Nest 5.20
The partitioner for the fluid dynamics kernel:

```
for (i = 0; i < medsiz; i++){
    record_dep(indx[0][i], i);    /*4 recording*/
    record_dep(indx[1][i], i);    /*5 recording*/}
```

Comparing with Hand-Written Partitioner As described before, procedure record_dep records data elements referenced in the current iteration and decides which

edges	sequential kernel	sequential partitioner	parallel kernel w. irregular partition	parallel kernel w. block partition
131072	8885	541	1392	2456
262144	18013	1001	2552	4360

Table 5.2: Running time in mini seconds of the fluid dynamics kernel on a 32-processor iPSC/860.

processors those referenced data elements and the current iteration can be mapped to. In order to compare our compiler-generated partitioner with the hand-coded partitioner from Joel Saltz *et al.* [27,95,115,114], we follow their partitioning strategy in generating code for record_dep. Due to dependence analysis, program slicing, RRE, statement substitution, and the same partitioning strategy, there is no difference between these two partitioners.

Comparing with Regular Partitioning Ir order to see that we really need a partitioner to reduce communication overhead, instead of partitioning the data regularly, we run the parallel code with two types of partitioning data. The parallel C code of the kernel is generated by the compiler designed and implemented by Berryman, Das, Hiranandani, Mavriplis, Saltz and Wu [27,95,115,114]. In the experiment, we use a 256 by 256 point mesh (data elements). Two sets of randomly selected edges (iterations) are used to connect those mesh points: one with 131072 edges and the other with 262144 edges. And we use block partition as the regular partition strategy.

The parallel code is executed on a 32-processor Intel iPSC/860 parallel machine, and the sequential kernel and partitioner are executed on a single iPSC/860 node. The timing of executing sequential and parallel codes for different sizes of input data with different partitions is described in Table 5.2.

From the timing, we know that, by paying the overhead for obtaining a better data layout, we can save more communication overhead, i.e. $(541 < 2456 - 1392)$

for size 131072 and $(1001 < 4360 - 2552)$ for size 262144. In real fluid dynamics simulation, the kernel loop will be iterated several times, which makes the scheduling overhead insignificant and a good data layout more critical.

The scheduling time is 6.1% (541/8885) for size 131072 and 5.6% (1001/18013) for size 262144 of the sequential execution time. And the scheduling time is 38.9% (541/1392) for size 131072 and 39.2% (1001/2552) for size 262144 of the parallel execution time. The overhead is also greatly amplified by parallelization.

Chapter 6

Conclusions

6.1 Summary of Contributions

This thesis has made the following contributions to the field of compiling high-level languages to massively parallel machines:

1. It presents a new dependence test with two independent contributions. First, the test includes in the system of dependence inequalities the information arising from a program's predicates. Second, it is an improved dependence test for equations with $-1, 0, 1$ coefficients. The test is more accurate for testing coupled array subscripts in statements with and without conditionals than previous dependence tests.

2. It presents a complete classification of static loop transformations by viewing the iteration space and statements as distinct dimensions in which code is restructured. From the classification, we have a clear picture of the previous techniques and how they can be extended to detect more parallelism.

3. It provides algorithms to find piecewise affine scheduling, which can detect more parallelism than unimodular loop transformations [10] including loop interchange, permutation and skewing.

4. It presents a new hybrid compiler/run-time method for scheduling and load balancing programs with extensive indirections or pointers. It demonstrates that compile-time analysis can help to make the run-time scheduling and data partitioning overhead for restricted nonaffine loops insignificant.

Massively parallel machine technology is the trend for building the fastest machines. Two software challenges must be overcome to make massively parallel machines truly usable: large-scale parallelism detection for fully utilizing the large number of processors, and good data layout for reducing communication overhead. The new compiler techniques presented in this thesis help to overcome these difficulties for two important classes of programs: those consisting of affine loops and those consisting of restricted nonaffine loops.

6.2 Directions for Future Research

The work presented in this thesis can be extended in many directions. We now describe some suggestions for future research.

Our new techniques are applicable to several well-known applications, e.g. Gauss-Jordan elimination, dynamic programming, transitive closure, shortest path, circuit simulation and fluid dynamics. More applications should be tested to see how general these techniques can be in practice for parallelizing real and complex programs.

For the static scheduler part, more research is necessary to find a general and efficient subdomain scheduling algorithm. We present an efficient heuristic to find subdomain schedules when subdomains are given, but we only find a general but complicated algorithm to obtain both the schedule and the subdomains in the same time.

For the dynamic scheduler part, more research can be done to reduce further the run-time scheduling overhead. Two possible directions are as follows:

The first direction is to investigate more types of redundant references for loop nests with indirections and pointers. Two types of redundant references discussed in this thesis are common references and subsumed references. There might be more types of references that are redundant and therefore can be eliminated from the run-time scheduler.

The second direction is to parallelize the run-time scheduler. The scheduler presented in this thesis is sequential. If the scheduler can be parallelized, then the run-time scheduling overhead can be further reduced and the memory problem can be circumvented. Our preliminary study shows that the IDG-constructor can be parallelized if the source loop only contains loop-invariant pointers (defined in Section 5.5.1), and parallel graph partitioning algorithms can be used to partition the IDG in parallel.

Bibliography

[1] H. Agrawal and J.R. Horgan. Dynamic program slicing. In *Proc. SIGPLAN '90 Conf. Program. Lang. Design and Implement.*, pages 246–256, 1990.

[2] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, 1974.

[3] J.R. Allen. *Dependence Analysis for Subscript Variables and Its Application to Program Transformation.* PhD thesis, Rice University, April 1983.

[4] J.R. Allen and K. Kennedy. Automatic loop interchange. In *Proceedings of the SIGPLAN'84 Symposium on Compiler Construction*, pages 233–246. ACM, 1984.

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

[6] U. Banerjee. Data dependence in ordinary programs. Master's thesis, University of Illinois at Urbana-Champaign, Nov. 1976.

[7] U. Banerjee. *Speedup of Ordinary Programs.* PhD thesis, University of Illinois at Urbana-Champaign, Oct. 1979.

[8] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

[9] U. Banerjee. A theory of loop permutation. Technical report, Intel Corporation, 1989.

[10] U. Banerjee. Unimodular transformations of double loops. In *Proc. 3rd Workshop on Programming Languages and Compilers for Parallel Computing.* UC. Irvine, 1990.

[11] S. Bhatt, M.C. Chen, C.Y. Lin, and P. Liu. Abstractions for parallel N-body simulation. In *Proc. Scalable High Performance Computing Conference*, pages 38–45, April 1992.

[12] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN'86 Symposium on Compiler Construction*, pages 162–175. ACM, 1986.

[13] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 53–65. ACM, 1990.

[14] D. Callahan and B. Koblenz. Register allocation via hierarchical graph coloring. In *Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 192–203. ACM, 1991.

[15] B. Chapman, P. Mehrotra, and H. Zima. VIENNA FORTRAN - A FORTRAN language extension for distributed memory multiprocessors. Technical Report 91-72, ICASE, September 1991.

[16] D.R. Chase, M. Wegman, and F.K. Zadeck. Analysis of pointers and structures. In *Proc. SIGPLAN '90 Conf. Program. Lang. Design and Implement.*, pages 296–310, 1990.

[17] M.C. Chen. A design methodology for synthesizing parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, Dec. 1986.

[18] M.C. Chen and Y. Choo. *Synthesis of a Systolic Dirichlet Product Using Non-Linear Domain Contraction*, pages 281–295. Elsevier Science Publishers B.V., 1989.

[19] M.C. Chen, Y. Choo, and J. Li. Compiling parallel programs by optimizing performance. *The Journal of Supercomputing*, 1(2):171–207, July 1988.

[20] Y. Choo and M.C. Chen. A theory of parallel program optimization. Technical Report TR-608, Yale University, Feb. 1988.

[21] *C* Reference Manual.* Thinking Machine Corp.

[22] *CM FORTRAN Reference Manual.* Thinking Machine Corp.

[23] W. Cook, A.M.H. Gerards, A. Schrijver, and E. Tardos. Sensitivity theorems in integer linear programming. *Mathematical Programming*, 34:251–264, 1986.

[24] R. Cytron. *Compile-Time Scheduling and Optimization for Asynchronous Machines.* PhD thesis, University of Illinois at Urbana-Champaign, Oct. 1984.

[25] R. Cytron, D.J. Kuck, and A.V. Veidenbaum. *The Effect of Restructuring Compilers on Program Performance for High-Speed Computers*, pages 39–48. Elsevier Science Publishers B.V., 1985.

[26] R. Das, R. Ponnusamy, J. Saltz, and D. Mavriplis. *Distributed Memory Compiler Methods for Irregular Problems – Data Copy Reuse and Runtime Partitioning*, pages 185–219. Elsevier Science Publishers B.V., 1992.

[27] R. Das, J. Saltz, D. Mavriplis, J. Wu, and H. Berryman. Unstructured mesh problems, Parti primitives and the ARF compiler. In *Parallel Processing for Scientific Computation, Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing, Houston TX*, April 1991.

[28] P. Debefve, F. Odeh, and A.E. Ruehli. *Waveform Techniques*, pages 41–127. Elsevier Science Publishers B.V., 1987.

[29] J.M. Delosme and I.C.F. Ipsen. Systolic array synthesis: Computability and time cones. Technical Report RR-474, Yale University, 1986.

[30] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[31] A. Dinning and E. Schonberg. An experical comparison of monitoring algorithms for access anomaly detection. In *Proc. of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 1–10, 1990.

[32] J. Fang and M. Lu. An iteration partition approach for cache or local memory trashing on parallel processing. In *Proceedings of the 4th Int'l Workshop on Languages and Compilers for Parallel Computing*, pages 313–327. Springer-Verlag, 1991.

[33] Z. Fang, P. Tang, P.C. Yew, and C.Q. Zhu. Dynamic processor self-scheduling for general parallel nested loops. *IEEE Trans. on Computers*, 39(7):919–929, July 1990.

[34] S.D. Feit. A fast algorithm for the two-variable integer programming problem. *Journal of the ACM*, 31(1):99–113, Jan. 1984.

[35] F. Fernandez and P. Quinton. Extension of Chernikova's algorithm for solving general mixed linear programming problems. Technical Report 437, INRIA-Rennes, Oct. 1988.

[36] J. Ferrante, V. Sarkar, and W. Trash. On estimating and enhancing cache effectiveness. In *Proceedings of the 4th Int'l Workshop on Languages and Compilers for Parallel Computing*, pages 328–343. Springer-Verlag, 1991.

[37] *FORTRAN 90 Standard.*

[38] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.W. Tseng, and M.Y. Wu. FORTRAN D language specification. Technical Report 90-141, Rice University, December 1990.

[39] M.R. Garey and D.S. Johnson. *Computers and Intracrability A Guide to the Theory of NP-Completeness.* 1979.

[40] G. Goff, K. Kennedy, and C.W. Tseng. Practical dependence testing. In *Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 15–29. ACM, 1991.

[41] C. Guerra and R. Melham. Synthesizing non-uniform systolic designs. In *Proceedings of the 1986 Int'l. Conf. on Parallel Processing*, pages 765–772. IEEE and ACM, 1986.

[42] W.L. Harrison. Compiling Lisp for evaluation on a tightly coupled multiprocessor. Technical Report 565, University of Illinois at Urbana-Champaign, March 1986.

[43] P.J. Hatcher, M.J. Quinn, A.J. Lapadula, B.K. Seevers, R.J. Anderson, and R.R. Jones. Data-parallel programming on MIMD computers. *IEEE Trans. on Parallel and Distributed Systems*, 3(2):377–383, July 1991.

[44] L.J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Trans. on Parallel and Distributed Systems*, 1(1):35–47, Jan. 1990.

[45] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C.W. Tseng. An overview of the FORTRAN D programming system. In *Proceedings of the 4th Int'l Workshop on Languages and Compilers for Parallel Computing*, pages 18–34. Springer-Verlag, 1991.

[46] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *Proc. SIGPLAN '89 Conf. Program. Lang. Design and Implement.*, pages 28–40, 1988.

[47] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graph. *ACM Trans. on Programming Languages and Systems*, 12(1):26–60, Jan. 1990.

[48] T.A. Johnson and D.J. Zukowski. Waveform relaxation based circuit simulation on the Victor (V256) parallel processor. Technical report, IBM, Sep. 1990.

[49] N.D. Jones and S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *9th ACM Symp. Principles Program. Lang.*, pages 66–74, 1982.

[50] Y.J. Ju and H. Dietz. Reduction of cache coherence overhead by compiler data layout and loop transformation. In *Proceedings of the 4th Int'l Workshop on Languages and Compilers for Parallel Computing*, pages 344–358. Springer-Verlag, 1991.

[51] R.M. Karp, R.E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, July 1967.

[52] D. Klappholz, K. Psarris, and X. Kong. On the perfect accuracy of an approximate subscript analysis test. In *Proc. 1990 Int'l Conference on Supercomputing*, 1990.

[53] D.E. Knuth. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms.* Addison-Wesley, 1968.

[54] X. Kong, D. Klappholz, and K. Psarris. The I test: A new test for subscript data dependence. In *Proceedings of the 1986 Int'l. Conf. on Parallel Processing.* IEEE and ACM, 1990.

[55] X. Kong, D. Klappholz, and K. Psarris. The I test: An improved dependence test for automatic parallelization and vectorization. *IEEE Trans. on Parallel and Distributed Systems*, 2(3):342–349, July 1991.

[56] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29:155–163, Oct. 1988.

[57] V.P. Krothapalli and P. Sadayappan. Removal of redundant dependences in DOACROSS loops with constant dependences. In *Proc. of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 51–60, 1991.

[58] D. Kuck, P.P. Budnik, S.C. Chen, D.H. Lawrie, R.A. Towle, R.E. Strebendt, E.W. Davis Jr., J. Han, P.W. Kraska, and Y. Muraoka. Measurements of parallelism in ordinary FORTRAN programs. *Computer*, 7(1):37–46, Jan. 1974.

[59] S. Kung. On supercomputing with systolic/wavefront array processors. *Proceedings of the IEEE*, 72(7):867–884, July 1984.

[60] S.Y. Kung, S.C. Lo, and P.S. Lewis. Optimal systolic design for the transitive closure and the shortest path problems. *IEEE Trans. on Computer*, 36(5):603–614, May 1987.

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

[62] J.R. Larus and P.N. Hilfinger. Detecting conflicts between structure accesses. In *Proc. SIGPLAN '88 Conf. Program. Lang. Design and Implement.*, pages 21–34, 1988.

[63] J.R. Larus and P.N. Hilfinger. Restructuring Lisp programs for concurrent execution. In *ACM/SIGPLAN PPEALS Parallel Program.: Exp. Appl. Lang. Syst.*, pages 100–110, 1988.

[64] P.Z. Lee and Z.M. Kedem. Mapping nested loop algorithms into multidimensional systolic arrays. *IEEE Trans. on Parallel and Distributed Systems*, 1(1):64–76, Jan. 1990.

[65] H.W. Lenstra. Integer programming with a fixed number of variables. *Math. of Operations Research*, pages 538–548, 1983.

[66] G.J. Li and Wah B.W. The design of optimal systolic arrays. *IEEE Trans. on Computer*, C-34(1):66–77, Jan. 1985.

[67] J. Li. *Compiling Crystal for Distributed-Memory Machines.* PhD thesis, Yale University, Oct. 1991.

[68] J. Li and M.C. Chen. Generating explicit communication from shared-memory program references. In *Proc. Supercomputing '90*, pages 865–876, 1990.

[69] J. Li and M.C. Chen. Index domain alignment: Minimizing cost of cross-reference between distributed arrays. In *Frontiers '90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, 1990.

[70] J. Li and M.C. Chen. Compiling communication-efficient programs for massively-parallel machines. *IEEE Trans. on Parallel and Distributed Systems*, 2(3), July 1991.

[71] J. Li and M.C. Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*, 1991.

[72] Z. Li. Intraprocedural and interprocedural data dependence analysis for parallel computing. Technical Report 910, University of Illinois at Urbana-Champaign, Aug. 1989.

[73] Z. Li and W. Abu-Sufah. On reducing data synchronization in multiprocessed loops. *IEEE Trans. on Computer*, C-36(1):105–109, Jan. 1987.

[74] Z. Li, P.C. Yew, and C.Q. Zhu. An efficient data dependence analysis for parallelizing compilers. *IEEE Trans. on Parallel and Distributed Systems*, 1(1):26–34, Jan. 1990.

[75] W.M. Lin and V.K.P. Kumar. A note on the linear transformation method for systolic array design. *IEEE Trans. on Computer*, C-39(3):393–399, March 1990.

[76] L.C. Lu. A unified framework for systematic loop transformations. In *Proc. of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 28–38, 1991.

[77] L.C. Lu and M.C. Chen. New loop transformation techniques for massive parallelism. Technical Report TR-833, Yale University, Oct. 1990.

[78] L.C. Lu and M.C. Chen. Subdomain dependence test for massive parallelism. In *Proc. Supercomputing '90*, pages 962–972, 1990.

[79] C. Mauras, P. Quinton, S. Rajopadhye, and Y. Saouter. Scheduling affine parameterized recurrences by means of variable dependent timing functions. In *Proceedings of the International Conference on Application Specific Array Processors*, Sep. 1990.

[80] S.P. Midkiff and D.A. Padua. Compiler algorithms for synchronization. *IEEE Trans. on Computer*, C-36(12):1485–1495, Dec. 1987.

[81] W.L. Miranker and A. Winkler. Spacetime representations of computational structures. In *Computing*, volume 32, pages 93–114, 1984.

[82] R. Mirchandaney, J. Saltz, R.M. Smith, D.M. Nicol, and K. Crowley. Principles of runtime support for parallel processors. In *Proc. 1988 ACM Int'l. Conf. Supercomput.*, pages 140–152, July 1988.

[83] D.I. Moldovan. On the analysis and synthesis of VLSI algorithms. *IEEE Trans. on Computers*, C-31(11):1121–1126, Nov. 1982.

[84] D.I. Moldovan. On the design of algorithms for VLSI systolic arrays. *Proceedings of the IEEE*, 71(1), 1983.

[85] D.I. Moldovan and J.A.B. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Trans. on Computers*, C-35(1), Jan. 1986.

[86] D. Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, Dec. 1986.

[87] C.D. Polychronopoulos and D.J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Trans. on Computers*, C-36(12):1425–1439, Dec. 1987.

[88] P. Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *Proceedings of 11th Annual Symposium on Computer Architecture*, pages 208–214, 1984.

[89] P. Quinton. Mapping recurrences on parallel architectures. In *Proceedings of the Third International Conference on Supercomputing*, pages 1–8. ICS, 1988.

[90] P. Quinton and V.V. Dongen. The mapping of linear recurrence equations on regular arrays. Technical Report 485, INRIA-Rennes, July 1989.

[91] S.K. Rao. *Regular Iterative Algorithms and their Implementations on Processor Arrays.* PhD thesis, Stanford University, Oct. 1985.

[92] M. Rosing, R. Schnabel, and R.P. Weaver. The DINO parallel programming language. Technical Report CU-CS-457-90, Univ. Colorado, Boulder, April 1990.

[93] J. Rowell. Prudential Securities pioneers supercomputing on Wall Street. *Supercomputing Review*, 5(4):24–26, April 1992.

[94] J.K. Salmon. *Parallel Hierarchical N-body Methods.* PhD thesis, Cal. Tech., 1991.

[95] J. Saltz, H. Berryman, and J. Wu. Runtime compilation for multiprocessors. *Concurrency: Practice and Experience*, 3(6):573–592, 1991.

[96] J. Saltz and M.C. Chen. Automated problem mapping: the Crystal run-time system. In *The Proceedings of the Conference on Hypercube Microprocessors, Knoxville, TN*, Sep. 1986.

[97] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *J. Parallel Distributed Comput.*, 8:303–312, April 1990.

[98] J. Saltz, R. Mirchandaney, and K. Crowley. Run-time parallelization and scheduling of loops. *IEEE Trans. on Computers*, 40(5):603–612, May 1991.

[99] H. Scarf. Integer programming and lattice theory. *Lecture Notes, Department of Computer Science, Yale University*, 1990.

[100] A. Schrijver. *Theory of Linear and Integer Programming. Wiley-Interscience series in Discrete Mathematics.* John Wiley and Sons, 1986.

[101] D.G. Shea, T.A. Johnson, and D.J. Zukowski. Joint study. *IBM T.J. Watson Research Center*, 1990-1991.

[102] Z. Shen, Z. Li, and P.C. Yew. An empirical study of FORTRAN programs for parallelizing compilers. *IEEE Trans. on Parallel and Distributed Systems*, 1(3):356–364, July 1990.

[103] J.P. Sheu and C.Y. Chang. Synthesizing nested loop algorithms using nonlinear transformation method. *IEEE Trans. on Parallel and Distributed Systems*, 2(3):304–317, July 1991.

[104] P. Tang, P.C. Yew, and C.Q. Zhu. Impact of self-scheduling order on performance of multiprocessor systems. In *Proc. 1988 ACM Int'l. Conf. Supercomput.*, pages 593–603, July 1988.

[105] M. Weiser. Program slicing. *IEEE Trans. on Software Engineering*, 10(4):352–357, July 1984.

[106] D.L. Whitaker and B. Grossman. Two-dimensional euler computations on a triangular mesh using an upwind, finite volume scheme. In *Proceedings AIAA 27th Aerospace Sciences Meeting*, Jan 1989.

[107] D.L. Whitaker, D.C. Slack, and R.W. Walters. Solution algorithms for the two-dimensional euler equations on unstructured meshes. In *Proceedings AIAA 28th Aerospace Sciences Meeting*, Jan 1990.

[108] M.E. Wolf and M.S. Lam. Maximizing parallelism via loop transformations. In *Proc. 3rd Workshop on Programming Languages and Compilers for Parallel Computing.* UC. Irvine, 1990.

[109] M.E. Wolf and M.S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):452–471, Oct. 1991.

[110] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, University of Illinois at Urbana-Champaign, Oct. 1982.

[111] M. Wolfe. Advanced loop interchanging. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 536–543, August 1986.

[112] M. Wolfe. More iteration space tiling. In *Proc. Supercomputing '89*, November 1989.

[113] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, 1989.

[114] J. Wu, J. Saltz, H. Berryman, and S. Hiranandani. Distributed memory compiler design for sparse problems. Technical Report 91-13, ICASE, Jan. 1991.

[115] J. Wu, J. Saltz, S. Hiranandani, and H. Berryman. Runtime compilation methods for multicomputers. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume 2, pages 26–30, 1991.

[116] C.Q. Zhu and P.C. Yew. A scheme to enforce data dependence on large multiprocessor systems. *IEEE Trans. on Computer*, C-36(6):726–739, June 1987.