Array Restructuring for Cache Locality

by

Shun-Tak Albert Leung

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

1996

Approved by [illegible]
(Chairperson of Supervisory Committee)

Program Authorized
to Offer Degree Computer Science and Engineering

Date July 22, 1996

University of Washington

Abstract

Array Restructuring for Cache Locality

by Shun-Tak Albert Leung

Chairperson of Supervisory Committee: Professor John Zahorjan
Department of Computer Science
and Engineering

Caches are used in almost every modern processor design to reduce the long memory access latency, which is increasingly a bottleneck to program performance. For caches to be effective, programs must exhibit good data locality. Thus, an optimizing compiler may have to restructure programs to enhance their locality. We focus on the class of restructuring techniques that target array accesses in loops.

There are two approaches to enhancing the locality of such accesses: *loop restructuring* and *array restructuring*. Under loop restructuring, a compiler adopts a canonical array layout but transforms the order in which loop iterations are performed and thereby reorders the execution of array accesses. Under array restructuring, in contrast, a compiler lays out array elements in an order that matches the access pattern, while preserving the flow of control. While loop restructuring has been studied extensively, array restructuring has received much less attention despite advantages such as its applicability to complicated loop structures that may hamper loop restructuring.

To fill the void, this dissertation investigates how to perform array restructuring effectively — efficiently, automatically, and generally. We present a formal framework for array transformations that meet these objectives. Such transformations are represented by linear transformations of array index vectors. Within this framework, we develop algo-

rithms to solve various problems in array restructuring: selecting transformations based on the access pattern, laying out elements of restructured arrays, and determining which elements are accessed by a loop and thus restructuring only that part of an array.

To evaluate our array restructuring technique, we implemented a prototype compiler and performed a series of experiments with loops commonly used in related loop restructuring studies. Experimental measurements showed that array restructuring improved performance substantially in many cases, despite a modest runtime overhead in some. Moreover, the results also indicated that array restructuring complemented loop restructuring in applicability and performance: it applied where loop restructuring did not; when both applied, it offered comparable, sometimes even better, performance; in cases where it did not perform as well, loop restructuring improved performance considerably anyway. This observation points to the potential benefit of integrating the two complementary approaches.

Table of Contents

List of Figures

List of Tables

List of Common Symbols

A	Access matrix
a	Access offset vector
B	Array bounds matrix
b	Array bounds vector
c	Center of symmetry of image polyhedron
$height(\ldots)$	Height of a matrix
i	Iteration vector
l_j	Lower array bound of the j-th dimension
$l_j(\ldots)$	Lower bound for the j-th dimension in the augmented bounds
m	Number of array dimensions
n	Number of loop levels in a loop nest
R	Transformation matrix for relaxed bounds
$rank(\ldots)$	Rank of a matrix
r_l	Lower bounds of relaxed bounds
r_u	Upper bounds of relaxed bounds
s	Scalar offset of an element into array
T	Index transformation matrix
t	Index transformation offset vector
u_j	Upper array bound of the j-th dimension
$u_j(\ldots)$	Upper bound for the j-th dimension in the augmented bounds
v	Linearization vector
x	Array index vector
y	Transformed array index vector

Acknowledgments

First of all, I must thank John Zahorjan, my advisor. Throughout the almost five years I have been working with him, he has offered not only advice and guidance, but also constant encouragement, support, and confidence in me — even, and perhaps especially, at those times when I have doubts. His optimism is contagious.

I would also like to thank Craig Chambers and Susan Eggers, members of my Reading Committee. Their valuable comments helped to improve this dissertation substantially.

National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign provided the Silicon Graphics Power Challenge used in some of my experiments. The SUIF compiler group of Stanford University provided the compiler infrastructure for my prototype. Ruth Anderson and Thu Nguyen installed the SUIF compiler locally and helped me with its use.

My parents, to whom this dissertation is dedicated, have always had my future in mind. Their silent support and sacrifice have enabled me to pursue my dreams and come thus far.

Last but certainly not least, praise and gratitude are due to God, who has showered me with undeserved blessings beyond what I dare to dream of.

Dedicated to my parents Sze-Wing Leung and Suet-Chor Kwan

Part I

Introduction

Chapter 1

Introduction

1.1 Motivation

As the gap between processor and memory speeds continues to widen, it is increasingly important to execution performance that the average latency of memory accesses be reduced. One proven hardware technique is the use of caches. Caches appear in virtually all major processor designs [Bannon and Keller 1995; Hunt 1995; Levitan, Thomas, and Tu 1995; Papworth 1996; Tremblay and O'Conner 1996; Yeager 1996]. The memory system is structured as a hierarchy of caches in the hope that most data accesses can be satisfied by levels that have short access times, thus avoiding long trips to relatively slow memory. This technique exploits the observation that many programs exhibit locality of access: soon after accessing a piece of data, the program tends to access the same data again (which is called temporal locality) or other nearby data (spatial locality).

The use of caches in turn influences programming practice. To achieve good performance, programs must utilize the cache effectively. Therefore, programmers conscious of the heavy cache miss penalty will try to write code that exhibits good data locality — both temporal and spatial. For example, they can make use of blocked algorithms, which seek to operate on data already in the cache as much as possible before bringing another set of data into the cache [Lam, Rothberg, and Wolf 1991]. While offering maximum performance, manual tuning is not only time-consuming, but also makes programs less portable across hardware platforms. A program hand-optimized for one architecture may perform much worse on another.

Compiler-directed locality optimizations promise high performance without sacrificing portability or demanding excessive programmer effort, although they may not outdo manual tuning. A compiler can analyze the data access pattern and restructure programs to enhance their locality, perhaps even tailoring them to the particular target architecture.

This dissertation focuses on compiler-directed program restructuring techniques that target array accesses in loops, which represent a significant fraction of the total execution time in many applications. There has been intense interest in such techniques. Some of the extensive research is surveyed in Chapter 9. These techniques have received much attention in part because array accesses in loops are often executed many times. Any effort spent on optimizing even a small number of them promises huge reward in execution performance. Their regularity also makes them especially amenable to compiler analysis.

1.1.1 Two Approaches to Improving Locality

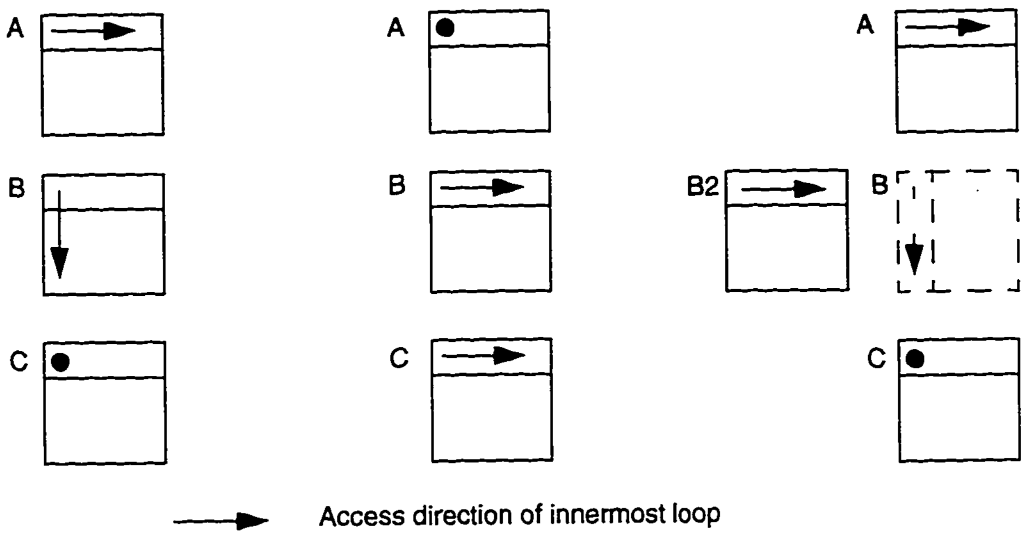
To improve the locality of array accesses in a loop, we can restructure the loop or restructure the arrays. In the *loop restructuring* approach, the compiler modifies the loop's control structure to change the execution order of the iterations (or parts of iterations) and thereby the sequence of array accesses. We might, say, bring iterations accessing the same array element closer together in execution order; this would improve temporal locality. In the *array restructuring* approach, the compiler changes the storage order of an array according to its access pattern. We might, say, store an array in column-major order if it is accessed column by column, thus enhancing spatial locality. Although previous work has concentrated on loop restructuring, in fact both approaches can enhance locality because locality results from an interaction between the execution order of accesses and the storage order of elements, rather than from either execution order or data layout alone. For concreteness, let us consider an example.

Figure 1.1 shows a straightforward implementation of a dense matrix multiplication as well as graphical representations of the access patterns for the three arrays involved. We show the array laid out in row-major order. We first examine the locality exhibited by the

```
C is initially zero
B is row-major
FOR i = 1, 100
 FOR j = 1, 100
  FOR k = 1, 100
   C[i,j] = C[i,j] +
     A[i,k] * B[k,j]
```

```
C is initially zero
B is row-major
FOR i = 1, 100
 FOR k = 1, 100
  FOR j = 1, 100
   C[i,j] = C[i,j] +
     A[i,k] * B[k,j]
```

```
C is initially zero
Copy B[x,y] to B2[y,x]
FOR i = 1, 100
 FOR j = 1, 100
  FOR k = 1, 100
   C[i,j] = C[i,j] +
     A[i,k] * B2[j,k]
```



(a) Original Array Layout (b) Loop restructuring: Interchange loops j, k (c) Array restructuring: B2 = transpose of B

Figure 1.1: A Simple Example — Matrix Multiply

original loop as shown in Figure 1.1(a). The innermost loop computes a single element of the result array C as the dot product of a row of A and a column of B.

- The accesses to `C[i,j]` have excellent temporal locality because the innermost loop reuses `C[i,j]` many times. Moreover, after scalar replacement [Bacon, Graham, and Sharp 1994] — a compiler optimization that replaces an array element with a scalar variable — the running sum is accumulated in a register; `C[i,j]` is read from memory only before the innermost loop and written back after.

- The access to `A[i,k]` has modest temporal locality and good spatial locality. The innermost loop reads a row of elements, which are consecutive in memory because

A is row-major. Thus, the innermost loop goes through adjacent memory locations, with good spatial locality. Furthermore, although iterations of the innermost loop read different elements, consecutive executions of the innermost loop reuse the same row. Temporal locality is moderately good, provided that a row of A is short enough to stay in cache until it is reused.

- The access to B[k,j] is problematic: it has poor temporal as well as spatial locality. Since the innermost loop goes through the row-major array column by column, consecutive iterations touch elements far apart in memory. These elements are in different cache lines (unless the arrays are unrealistically small). Each access therefore potentially results in a cache miss. Moreover, temporal locality is poor because the loop nest goes through the entire array before accessing any element again, by which time the cache line containing the reused element has most likely been displaced from the cache by the huge number of intervening accesses.

Loop restructuring improves the problematic access to B, as well as overall performance, despite some side-effects on the other two arrays. In part (b) of the figure, we interchange the two inner loops[1]. This alters the execution order of the iterations. For this loop nest, such a change respects all existing loop-carried dependences and therefore preserves program semantics. Under the new execution order, the innermost loop goes through array B row by row. Spatial locality is much better than before, although temporal locality remains unchanged because array B is still read in whole before being reused. The new execution order also affects accesses to arrays A and C. In particular, the innermost loop now reuses the elements of A and updates C row by row. Experiments have shown that these changes together lead to substantially better overall performance [Kennedy and McKinley 1992].

1. This simple loop nest could be improved further using loop tiling [Lam, Rothberg, and Wolf 1991; Wolf and Lam 1991a]. For simplicity, we present this illustrative example with loop interchange only.

Array restructuring also improves the problematic access to B, but without affecting the other accesses or temporal locality. Figure 1.1(c) illustrates this. While the loop structure is unchanged, the loop accesses a newly defined array B2, instead of the original array B. B2 contains the same elements as B, but in a different order determined by the access pattern: in this case B2 is the transpose of B. Thus, the access to B[k,j] is replaced by an access to the corresponding element in B2, namely B2[j,k]. As a result, the innermost loop goes through B2 row by row. This is logically equivalent to going through B column by column but has far better spatial locality. In addition to transforming the accesses to B, the compiler also generates code to copy data from B to B2 before loop execution[2]. If the array were updated, the data would also have to be copied back afterward. Notice that array restructuring has not changed temporal locality: the loop nest in Figure 1.1(c) still accesses the entire array B2 before reusing any element. Finally, the other two arrays are accessed in exactly the same way as before because the iteration execution order has not changed.

1.1.2 Comparing Array and Loop Restructuring

While we can often improve locality by either loop or array restructuring, each approach has distinct advantages as well as disadvantages, making one or the other more appropriate in a given situation.

First, array restructuring can be easily applied to complicated loop structures that sometimes hamper, if not frustrate, loop restructuring. This is because array restructuring does not change a program's flow of control, only its data layout. By contrast, loop restructuring is harder to apply because it affects control flow. Reordering data accesses without altering what the program computes requires careful compiler analysis to guarantee that the new order respects the same dependences as the old under all circumstances.

2. The original array B, and the data copying between B and B2, can be eliminated if all accesses to B throughout the program are replaced by accesses to B2. We discuss this possibility in Section 3.3.4.

This is especially difficult for complicated loops such as those that are imperfectly nested, contain conditional statements, have complex loop-carried dependences, have dependences dependent on runtime data, and so on. For example, while some of the most sophisticated loop restructuring techniques may be able to transform an imperfect loop nest implementing LU decomposition without pivoting [Anderson, Amarasinghe, and Lam 1995; Wolf and Lam 1991a], they would be frustrated by pivoting since the loop-carried dependences are determined by data available only at run time. Insufficient or imprecise compile-time information may also prevent the compiler from applying loop transformations desired for locality.

Second, array restructuring, by definition, affects spatial locality but not temporal locality, whereas loop restructuring affects both. Temporal locality is the property that after accessing a data item, programs tend to access it again soon, while spatial locality concerns the likelihood of accessing nearby data. Array restructuring does not affect temporal locality because temporal locality relates to how soon the same data item is accessed again, which in turn depends on the order of all accesses but not where that data item is placed in memory[3].

This property of array restructuring may be viewed as a limitation but can also be exploited to advantage. Since array restructuring does not affect temporal locaiity, it obviously cannot be used to improve temporal locality. On the other hand, we can use array restructuring to improve spatial locality *without jeopardizing temporal locality.* With loop restructuring, however, we may have to strike a balance between the improvement in spatial locality won by a loop transformation and any accompanying degradation in temporal locality.

3. *Locality* is defined here as a property of the access pattern and data layout. We distinguish it from *reuse*, especially reuse by a specific level of the memory hierarchy. Whether reuse results from locality depends on architectural parameters such as cache size, cache line size, replacement policy, etc. While temporal locality is not affected by array restructuring, the degree of temporal reuse may.

Third, when a loop accesses more than one array, array restructuring can transform each individually in the best way, independently of how, or whether, the others are restructured. Not only does this simplify analysis, but it also avoids the dilemma confronted in loop restructuring that the locality for one array may be improved only at the expense of another. By contrast, loop restructuring inevitably affects all the loop's accesses and thus may necessitate such a tradeoff. In the earlier example, array restructuring improves the access to array B without affecting those to A and C. In contrast, with loop restructuring, we must assess how the loop transformation impacts accesses to all three arrays before concluding that it will likely increase overall performance.

Analogously, when an array is accessed by more than one loop, array restructuring affects all these loops, complicating analysis and sometimes resulting in a painful tradeoff, while loop restructuring simply considers each loop independently. However, array restructuring may still avoid such tradeoffs between loop nests if arrays are restructured dynamically in between. The runtime cost must, of course, be considered. Section 3.3.4 explores this further.

1.1.3 Understanding Array Restructuring

We have seen that both loop and array restructuring can improve locality of array accesses in loops, and that each approach has its advantages as well as disadvantages. Many of them are "symmetric" to each other, making one or the other approach more suited to a given case.

However, relatively little attention has been given to array restructuring; most research effort so far has focused on loop restructuring techniques, including both individual transformations and, more recently, unifying analysis frameworks (such as one using nonsingular matrices [Li and Pingali 1993c]). To facilitate comparison with our work, we defer the discussion of these contributions to Chapter 9, which surveys the extensive, long-standing work on loop restructuring and some recent progress in array restructuring.

This dissertation fills the void in the understanding of array restructuring. We aim to identify, understand, and address issues that must be dealt with in order to perform array restructuring effectively — efficiently, automatically, and as generally as possible. The insights will help us not only to perform array restructuring *per se* better, but also to integrate both forms of restructuring for maximum effectiveness.

1.2 Contributions

We make the following contributions in this dissertation:

- We develop a formal framework for performing array restructuring efficiently, automatically, and as generally as possible.
- We devise detailed algorithms for various steps in the array restructuring process within this framework and show that they solve the problems they address.
- We demonstrate the feasibility of array restructuring by implementing a prototype compiler incorporating these algorithms.
- We evaluate our array restructuring techniques through a series of experiments using loops from benchmarks and the related literature.

1.3 Organization

The rest of this dissertation is organized as follows. Part II presents the analysis techniques to restructure arrays. In particular, Chapter 2 presents our framework for implementing array restructuring without incurring extra indexing overhead. It is based on the linear transformation of array index vectors between the original and restructured arrays. Chapter 3 describes our algorithms for computing an index transformation to improve locality based on the data access pattern. Chapter 4 discusses how to lay out the elements of the restructured array in such a way that finding any element from its array indices is as efficient as in the conventional (untransformed) case, a nontrivial problem because of the generality of our array transformations. These three chapters assume that arrays are restructured in whole. Chapter 5 presents techniques to restructure only elements that are accessed by a given loop and to calculate such a subset of elements.

Part III presents our results. Chapter 6 outlines the implementation of a prototype compiler for array restructuring. Chapter 7 reports a series of experiments to evaluate our array restructuring techniques and study how they compare and interact with existing loop restructuring techniques. Chapter 8 presents experimental results for parallel execution.

Finally, Part IV concludes this dissertation by placing it in a larger context. Chapter 9 reviews related work. Chapter 10 summarizes our contributions and suggests future research directions.

Part II

Analysis

Chapter 2

Array Restructuring Framework

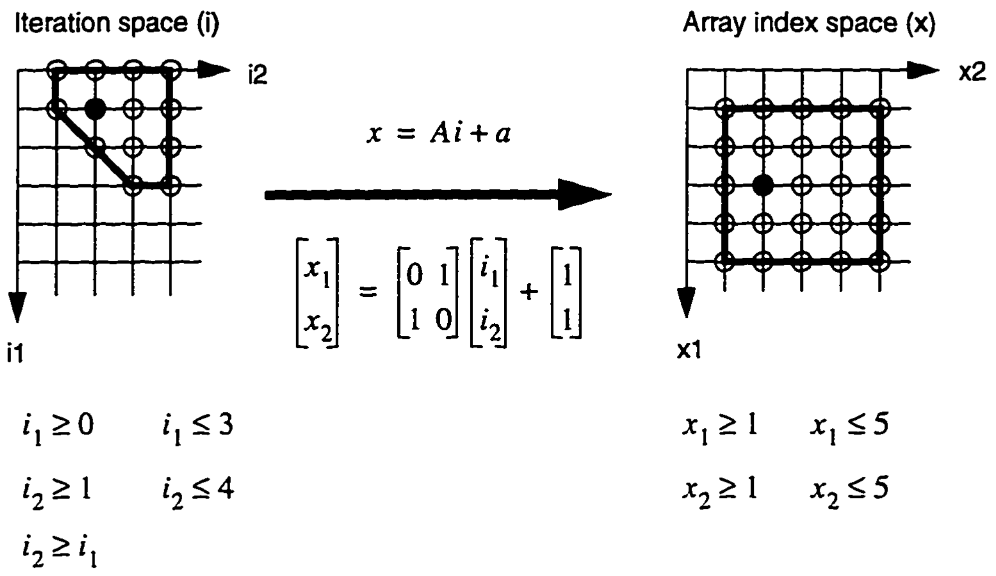
To automate array restructuring, we need a concrete framework for representing and deciding how arrays are structured. This chapter presents the framework used in our analysis. First, we outline a loop restructuring framework based on linear algebraic representations of iterations, array elements, and accesses. Given this background, we present our array restructuring framework and discuss how it affects the generated code. Finally, we compare this framework with several other alternatives to show the rationale behind our choice.

2.1 Loop Restructuring Framework: A Tutorial

Arrays are restructured for better locality based on how they are accessed by the loop. To decide automatically how each array should be restructured, we must first represent the abstract notion of "access pattern" in a concrete way that allows formal analysis. For this purpose, we use a linear algebraic framework that others have found useful for analyzing certain loop transformations and their effects on array accesses [Li and Pingali 1993c; Wolf and Lam 1991b]. A tutorial is given here for completeness. Using the illustrative example in Figure 2.1, we first discuss the representations of iterations and array elements and finally those of array accesses.

Consider an n-deep perfect loop nest such as the one at the top of Figure 2.1, for which $n = 2$. We assume that all the loop variables are incremented at unit stride. Loop nests that do not meet this condition are first normalized to the canonical form by adjusting the loop bounds.

```
Declare X[1:5,1:5]
FOR i1 = 0, 3
 FOR i2 = max(i1,1), 4
  X[i2+1,i1+1] = ...
```



$$x = Ai + a$$

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$i_1 \geq 0 \quad i_1 \leq 3$

$i_2 \geq 1 \quad i_2 \leq 4$

$i_2 \geq i_1$

$x_1 \geq 1 \quad x_1 \leq 5$

$x_2 \geq 1 \quad x_2 \leq 5$

Figure 2.1: Linear Algebraic Representation of Array Access

The left part of Figure 2.1 illustrates how iterations are represented. Each iteration can be uniquely identified by an n-dimensional *iteration vector*, denoted i, whose components are the values of the loop variables (ordered from the outermost to the innermost) for that iteration. The *iteration space* is an n-dimensional vector space that contains the iteration vectors. Clearly, iteration vectors must be integral because the loop variables can assume only integer values. However, not every integral vector identifies an iteration.

The loop bounds determine which integral vectors in the iteration space correspond to iterations. In Figure 2.1, for instance, only the grid points marked by circles identify iterations. We can represent the loop bounds algebraically as well as geometrically. Assume that the lower (upper) bound for the k-th loop variable, i_k, is the maximum (minimum) of one or more affine functions[1] of the enclosing loop variables i_1, i_2, ..., i_{k-1}. Under this assumption, we can represent the loop bounds algebraically by a conjunctive set of linear inequalities involving components of the iteration vector: each affine function in the bounds for i_k gives rise to a linear inequality involving i_1, i_2, ..., i_k. In Figure 2.1, the inequalities derived from the loop bounds are shown below the iteration space. Geometrically, the loop bounds are represented by a convex polyhedron[2] in the iteration space bounded by hyperplanes corresponding to the inequalities. As shown in Figure 2.1, the bounds for the example loop nest form a pentagon. Only iteration vectors whose components satisfy the inequalities (or, equivalently, integral points lying within the polyhedron) identify iterations of the loop.

Since all the loops have unit stride, loop execution may be viewed as enumerating the valid iteration vectors in lexicographic order[3] or, equivalently, the grid points in the polyhedron (the circles in Figure 2.1) in row-major order. The corresponding iterations are performed in that order.

Array elements can be represented in a similar way, as illustrated by the right part of Figure 2.1. For each array, an element is identified by an *index vector* comprising the individual array indices for the element. The *index space* for that array is the vector space containing the index vectors. The array bounds determine the set of valid index vectors — integral vectors in the index space that truly identify array elements. These bounds are represented by a conjunctive set of linear inequalities of the array indices or, equivalently, a convex polyhedron in the index space. In fact, because the lower and upper bounds for all array indices are constants, all the inequalities involve a single array index and the convex polyhedron is simply a rectilinear region. Under this framework, a row-major storage

1. In this context, an affine function is simply a linear combination of the arguments plus a constant, such as $f(x, y) = 2x + 3y + 4$.

2. A polyhedron is convex if and only if the line segment between any two points in the polyhedron also lies entirely in the polyhedron.

3. A vector x is lexicographically after, or greater than, another vector y if and only if the first nonzero component of $x - y$ is positive. The definition for "lexicographically before" or "less than" is analogous.

order means that array elements are laid out in lexicographic order of their index vectors: the element identified by index vector x is stored before (not necessarily immediately before) the one identified by x' if and only if x is lexicographically less than x'.

Given these representations of iterations and array elements, we can now represent an array access with a mapping from the iteration space to the index space: the mapping takes iteration vectors to the index vectors of those array elements accessed by the corresponding iterations. We focus on the case where all the array indices are affine functions of the loop variables. (More general access patterns will be discussed in Section 3.3.2.) In this case, which is illustrated in Figure 2.1, the mapping itself is also affine and thus can be written as

$$x = Ai + a \tag{2.1}$$

where i is the iteration vector, x the index vector for the accessed element, A a constant matrix containing the coefficients of the (affine) index expressions for this access, and a a constant vector containing the offsets in those expressions.

The matrix A is called the *access matrix*. It holds the key access pattern information used later in our analysis. For an access to an m-dimensional array inside an n-deep loop nest, the access matrix has m rows and n columns. Each row corresponds to an array index expression whereas each column corresponds to a loop variable. From the viewpoint of an array index, corresponding rows of the access matrix A and the offset vector a specify how the array index is computed from the loop variables. For example, the first rows of A and a in the figure express the fact that the first array index is the inner loop variable (`i2`) plus 1.

While it is natural to interpret the meaning of A row by row, each column also gives key information. From the viewpoint of a loop variable, the corresponding column of A indicates how the array indices change as the loop variable is incremented. The last column of A, for instance, indicates that when the inner loop variable is incremented, the first array index increases by one while the second is unchanged.

2.2 Array Restructuring Framework

We now present our array restructuring framework and discuss its impact on the generated code. Its basic idea has already been introduced in the matrix multiplication example in Section 1.1.1. Given the linear algebraic framework just described, we can now express it more formally, thus paving the way for the analysis to be presented later. As a convention throughout this dissertation, all arrays are presented as in row-major order.

The following discussion refers to Figure 2.2, which uses the same example as Figure 1.1 on page 4 except for the renaming of loop variables. We focus on the array being restructured — array `B`. The upper half of the diagram shows the situation in the original loop, whereas the lower half demonstrates the effects of restructuring the array `B`. The original and transformed codes are shown at the top and bottom respectively.

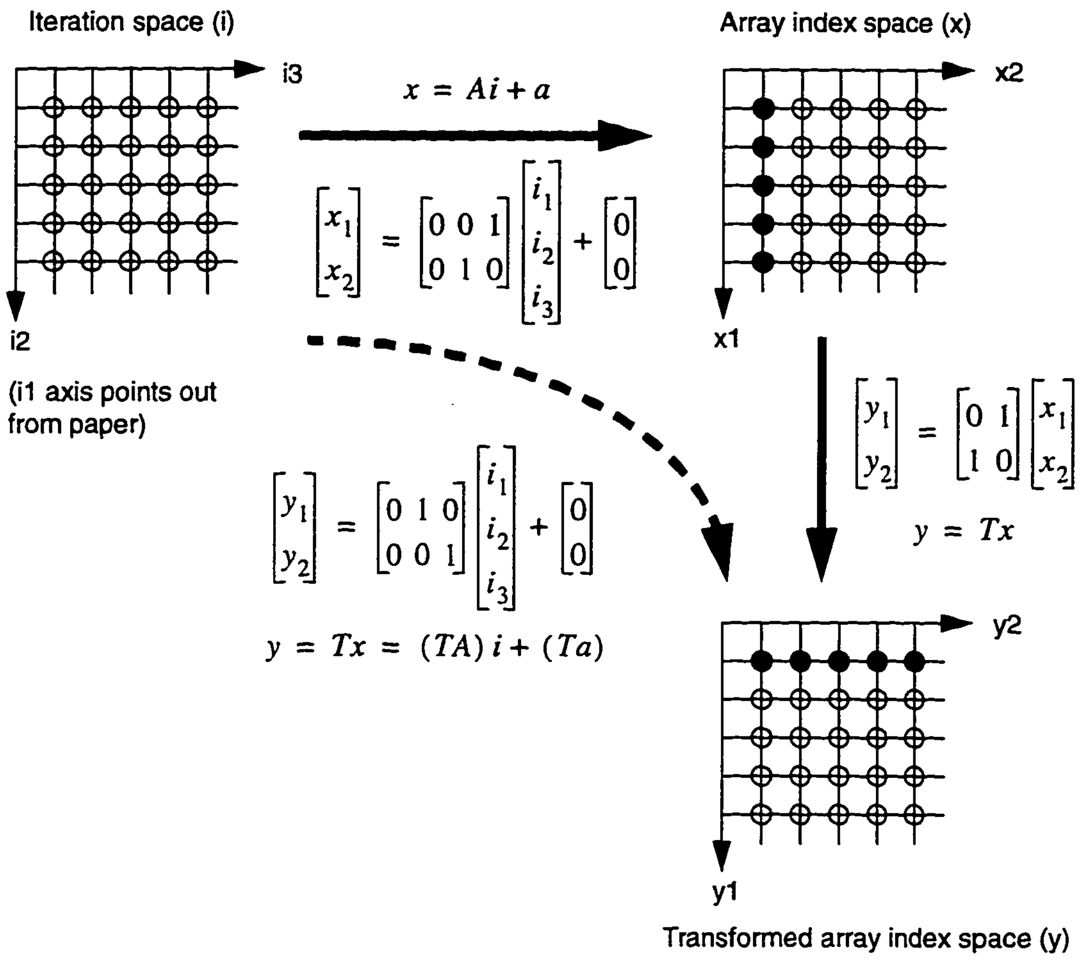
To restructure an array, we define another array with the same number of dimensions. Let us call this new array the *restructured array.* In general, the restructured array only needs to contain those elements of the original array that are actually accessed by the loop nest in question. We discuss this fully in Chapter 5. For the moment, however, we require that the restructured array contains the same elements as the original array, but in a different order.

Corresponding elements of the original and restructured arrays are related by a linear transformation of index vectors[4]. In other words, at each array access, instead of using the original array indices (components of the index vector x) to find an element in the original array, we apply a linear transformation to x to obtain a *transformed index vector* y and then use y to find the corresponding element in the restructured array. In equation form, we can write

$$y = Tx \tag{2.2}$$

4. Section 2.3 explains why affine transformations of index vectors, which are even more general than linear transformations, are not used.

```
FOR i1 = 1, 100
 FOR i2 = 1, 100
  FOR i3 = 1, 100
   C[i1,i2] = C[i1,i2] +
    A[i1,i3] * B[i3,i2]
```


Iteration space (i)
i3
i2
(i1 axis points out from paper)
$x = Ai + a$
$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \\ i_3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$
Array index space (x)
x2
x1
$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$
$y = Tx$
$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \\ i_3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$
$y = Tx = (TA)i + (Ta)$
y2
y1
Transformed array index space (y)

```
Copy elements of B to B2
FOR i1 = 1, 100
 FOR i2 = 1, 100
  FOR i3 = 1, 100
   C[i1,i2] = C[i1,i2] +
    A[i1,i3] * B2[i2,i3]
```

Figure 2.2: Array Restructuring by Linear Transformation of Index Vector

where T is the transformation matrix that defines the linear transformation applied to the original index vector and thus also defines the correspondence between elements of the original and restructured arrays. Note that T must be nonsingular[5] so that each transformed index vector y corresponds to only one original index vector x. Otherwise, multiple elements of the original array would be assigned to the same element in the restructured array. Further conditions on T are imposed in Section 3.1.3. For example, in Figure 2.2 the matrix $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ expresses the fact that array `B2` is the transpose of array `B`: the linear transformation it defines maps a column of elements in the original array to a row of elements in the restructured array (both sets of elements are marked by filled circles in the figure). Naturally, array transpose is only a special case; the use of linear transformation allows much more general forms of array restructuring.

Despite the extra index vector transformation, this array restructuring framework does not involve extra indexing overhead at each array access. In other words, it is no less efficient to find an element in the restructured array than in the original array. This is because the transformed index vector y can be computed directly from the iteration vector i in the same way as the original index vector x, without explicitly applying the transformation to x and thus without incurring the associated overhead.

To see this, let us look at Figure 2.2 again. Iteration i accesses an original array element with index vector $x = Ai + a$. After array restructuring, it accesses a restructured array element with transformed index vector $y = Tx$, which can be expressed directly in terms of the iteration vector i as

$$y = Tx = T(Ai + a) = (TA)i + (Ta) \qquad (2.3)$$

Thus, the transformed index vector y, like the original x, is an affine function of the iteration vector i, although the two functions may involve different constant matrices and vectors (TA and Ta versus A and a). A compiler can apply the same optimization technique

5. A matrix is nonsingular if and only if it is square and its determinant is nonzero.

in both cases: integer multiplications and additions required for computing an element's address from its array indices are strength-reduced to pointer increments at various levels of the loop nest [Aho, Sethi, and Ullman 1986].

Apart from the transformed array accesses, few other changes to the generated code are required. Neither the loop bounds nor the loop nesting structure need be modified; the flow of control through the loop nest remains unchanged. Since we restructure arrays dynamically at run time, the program may have to copy elements from the original to the restructured array before loop execution and, if the array is updated, copy elements back to the original afterward. The compiler must generate code to perform these operations.

While code generation is relatively straightforward, choosing the transformation matrix T is far from trivial. Also, laying out elements of the restructured array (i.e., defining how the transformed index vector y is mapped to an element's actual location) is a complicated question with the full generality of our framework. The next two chapters are devoted to these two issues.

2.3 Alternatives

We have presented our array restructuring framework. Corresponding elements of the original and restructured arrays are related by a linear transformation of index vectors. In this section, we discuss the rationale behind this choice by comparing it with several other plausible alternatives.

The comparison is based on three criteria.

- **Generality**. The framework should be general, but should not incur an unnecessary cost. More specifically, we would like a framework that can represent as general relationships as possible between elements of the original and restructured arrays. We consider whether it subsumes less general options. However, this generality should not come at the price of extra overhead in the simple cases where the subsumed options would suffice.

- **Indexing overhead.** We wish to minimize the indexing overhead (i.e., the runtime cost of finding an element) incurred by array restructuring. Any increase in this cost may substantially impact performance because it exacts a penalty every time the restructured array is accessed.

- **Indexing data structures.** We would like to avoid using indexing data structures — auxiliary data structures for locating array elements — as much as possible. They are undesirable for several reasons. First, they take up memory. Second, accesses to these data structures may degrade performance, especially if they cause cache misses. A third and related reason is that these accesses compete with accesses to "real" data for space in the cache since the two kinds of accesses are likely to be interspersed.

We now consider four options. We describe each option, assess it using the three aforesaid criteria, and explain why it is or is not selected.

The first option is to permute the array dimensions. For example, the restructured array element `X2[i,j,k]` corresponds to the original array element `X[j,i,k]` for any values of `i`, `j`, and `k`. In two dimensions, this reduces to an array transpose because the only possible permutation is an exchange. Permutation has been used in previous work to improve data locality and reduce false sharing [Ju and Dietz 1992; Anderson, Amarasinghe, and Lam 1995]. This transformation is the least general of the options considered in this section. It is a straightforward extension of the traditional row-major and column-major storage orders and requires no more indexing overhead than these cases. Nor does it require any indexing data structures to locate array elements.

The second option is the one we have chosen — linear transformation of index vectors. It is much more general than permutation, subsuming the latter as a special case where the transformation matrix T is a permutation matrix[6]. The example in Figure 2.2 is one such case. As discussed earlier, applying a linear transformation to index vectors requires no more indexing overhead than what is traditionally required. Also, like permutation, it requires no indexing data structures. Therefore, the generality does not carry an unnecessary cost even when simple permutation would have sufficed. In fact, the linear transformations selected automatically in such cases would correspond to permutations anyway. The extra generality is especially useful if accesses go through the array index space diagonally (or in any direction beyond just the natural orthogonal ones), as in banded matrix computations. Unlike permutation, however, this option allows an infinite number of possible transformations. Therefore, we need algorithms to choose one that achieves our goal; a brute force method that evaluates possibilities exhaustively is unacceptable. These algorithms will be discussed later. In short, we choose this option because it provides an extra level of generality that is useful in some cases but carries no extra runtime overhead. We now discuss two other options and explain why they are rejected.

The third, even more general option is to apply an affine, rather than linear, transformation to index vectors. In other words, the transformed index vector y is related to the original index vector by $Tx + t$, with an additional offset vector t. This subsumes the linear case with a zero t. Like the previous two options, this one does not need indexing data structures. Nor does it add to the indexing overhead because, as for linear transformation, the transformed index vector y can be expressed directly in terms of the iteration vector i with an affine function:

$$y = Tx + t = T(Ai + a) + t = (TA)i + (Ta + t) \qquad (2.4)$$

The only difference lies in the offset vector — $Ta + t$ versus Ta in the linear case. Despite its "free" generality, this option is not selected because the generality has no effect in practice. A different offset vector changes neither the storage order of the elements nor the memory access pattern. It merely shifts the positions of all the elements by the same

6. All rows and columns of a permutation matrix have exactly one nonzero element, which is 1. Multiplying a permutation matrix to a vector in effect permutes the vector's components.

distance in memory. In other words, there is no practical reason why we should choose t to be a nonzero rather than a zero[7]. The latter case simply reduces to a linear transformation.

Finally, let us consider the completely general mapping. Implementing this requires the use of indirection arrays. `X[i]` of the original array is stored in `X2[IDX[i]]` of the restructured array, where `IDX` is itself an array with one element for each element of `X`. This is the most general option conceivable since it allows array elements to be stored in an arbitrary order defined by the indirection table `IDX`. However, it also leads to significant indexing and memory overheads because of its need for indexing data structures. Each access to `X` is replaced by two accesses: one to `IDX` and one to `X2`. Worst of all, the access pattern for `IDX` is identical to that for the original array `X`. Therefore, if the latter has poor locality, so does the former: if an access causes a cache miss in the original array, it would also cause a cache miss in the identically laid out indirection array[8]. Therefore, despite its full generality, we decide against the use of indirection tables.

2.4 Summary

We have presented our formal framework for performing array restructuring. A restructured array is defined to contain the same elements as the original array in a different order. Finding an element in the former requires, conceptually, applying a linear transformation to the original index vector. However, no extra indexing overhead is actually involved because the linear transformation need not be applied explicitly. Instead, we can directly compute the transformed index vector from the iteration vector in the same way as

7. Shifting an entire array in memory may matter when we consider conflict misses arising from accesses to different data structures, particularly arrays. We consider this part of a broader question: relative placement of different data structures. As far as accesses to the array in question are concerned, however, shifting has no practical effect.

8. Indirection tables have been successfully used to reduce false sharing on shared-memory multiprocessors [Jeremiassen 1995; Jeremiassen and Eggers 1995]. They can be effective in this context because misses in the original array, which is written, are coherence misses caused by false sharing and hence would not appear in the indirection array, which is only read, even if both arrays are laid out and accessed identically.

we compute the original one. We have also considered several other alternatives that we have rejected in favor of linear index transformations.

Chapter 3

Choosing an Index Transformation

The index transformation matrix T defines how an array is restructured. This chapter addresses the key question of choosing T based on the access pattern to improve locality. We start in Section 3.1 with a set of requirements on the matrix. Section 3.2 discusses how to choose T to meet these requirements. To begin with, we focus on a simple case: a single array access with only affine index expressions in a single perfect loop nest. More general access patterns are covered in Section 3.3.

3.1 Index Transformation for Better Locality

We have the following two requirements for the index transformation matrix T. How to find the matrix T is discussed later in Section 3.2.

- Given the access matrix A, the transformed access matrix TA should be "lower-triangular," in a loose sense to be defined later. This requirement follows from our goal of improving the spatial locality of the array access. We explain this fully in the first two subsections below, starting from the impact of the innermost loop and then considering the enclosing loops as well.
- T must be unimodular. We elaborate on this in the final subsection.

3.1.1 Innermost Loop

A suitable index transformation can improve spatial locality if it causes the innermost loop to go through memory consecutively when accessing elements in the restructured array[1].

If this cannot be achieved, at least we want the stride to be as small as possible. We now show what this means for the transformation matrix T.

We illustrate the following discussion with the example in Figure 3.1. The original loop is adapted[2] from the literature on loop restructuring [Li 1995; Li and Pingali 1993c]. It performs a symmetric rank-2k update[3] for banded matrices. Array restructuring transforms the original version shown at the top to the version below. Although the loop is shown in its entirety for completeness, in this discussion we need to consider only the highlighted access to array `X`. The other arrays can be restructured independently. (As for the other access to `X`, we will discuss how to handle it in Section 3.3.1.) The rest of Figure 3.1 gives the geometrical and algebraic representations of the highlighted access before and after array restructuring.

In the original loop, the highlighted access has poor spatial locality. As the innermost loop is executed, each successive iteration reads an array element in a different row and a different column because incrementing the variable `k` throughout the loop changes both the row and column indices of the array (the former by -1 and the latter by 1). Elements accessed by consecutive iterations are scattered far apart in memory, almost assuredly in different cache lines. Each successive access potentially results in a cache miss. Note that in this case simply transposing the array offers little help because the elements accessed by the innermost loop lie in different rows *and* different columns.

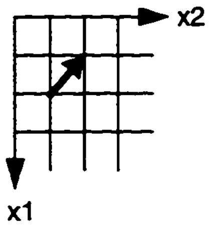
We can also view the problem geometrically, as shown by the middle part of Figure 3.1. The diagonal arrow indicates how the innermost loop goes through the array's index space: each successive iteration moves one row up and one column to the right, cor-

1. We assume, of course, that the innermost loop does not simply reuse the same array element. If it does, there is no reason to change this behavior. Nor is such a change possible by means of array restructuring.

2. The loop nest as presented in the literature uses Fortran-style arrays, with column-major storage and indices starting at 1 [Li 1995; Li and Pingali 1993c]. We have adapted the code for row-major arrays with zero lower bounds to conform with our convention of row-major storage and to simplify the bound and index expressions for presentation. The memory access pattern is nonetheless preserved.

3. A symmetric rank-2k update, part of the Basic Linear Algebra Subprograms Level 3 (BLAS3) [Dongarra et al. 1990], performs the matrix operation $Z = Z + X^tY + Y^tX$ on a symmetric matrix Z.

```
FOR i = 0, n-1
 FOR j = i, min(i+2*b, n-1)
  FOR k = max(i-b,j-b,0), min(i+b,j+b,n-1)
   Z[j-i,i] += X[j-k+b,k] * Y[i-k+b,k] +
               X[i-k+b,k] * Y[j-k+b,k]
```



$$A = \begin{bmatrix} 0 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix}$$

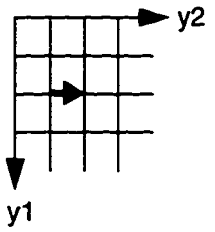
$$x = Ai + a$$

$$\downarrow$$

$$T = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

$$y = Tx = (TA)i + (Ta)$$

```
FOR i = 0, n-1
 FOR j = i, min(i+2*b, n-1)
  FOR k = max(i-b,j-b,0), min(i+b,j+b,n-1)
   Z[j-i,i] += X2[j+b,k] * Y2[i+b,k] +
               X2[i+b,k] * Y2[j+b,k]
```



$$TA = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 3.1: Improving Locality through Index Transformation

responding to the changes in the two array indices when loop variable k is incremented. The nonzero projection of this direction in the x_1 (vertical) dimension implies that successive iterations access elements in different rows. This, as we have seen, leads to poor spatial locality. Transposing the array merely reverses the direction, which remains diagonal and therefore equally bad for locality.

Finally, we can view the problem algebraically in terms of the nonzero structure of the access matrix, shown on the right in Figure 3.1. The last column of the access matrix represents changes in the two array indices as the innermost loop variable k is incremented through the innermost loop. This column is also the algebraic representation of the diagonal vector discussed above. Since the column starts with a nonzero (specifically -1), the first array index (i.e., the row index) changes with variable k. Again, this means that the innermost loop reads elements in different rows. Transposing the array would in effect interchange the two rows of the access matrix. Since the other element of the last column is also nonzero, this transformation does not solve the problem.

Our goal is to improve the spatial locality of the highlighted access. This goal can be stated in the same three ways we have just described the problem, and is met by the code, access matrix, and geometrical representation in the lower half of Figure 3.1. The array elements are laid out in such an order that in the transformed code, the innermost loop reads restructured array elements that lie in the same row and hence are stored contiguously. The innermost loop can thus go through memory consecutively, with better spatial locality than it has now. Geometrically, this means the innermost loop's access direction (the arrow in the figure) should be horizontal, rather than diagonal; in other words, it has no projection in the x_1 (vertical) dimension. Algebraically, the last column of the access matrix should start with a zero. In this way, the first array index (i.e., the row index) does not change with the increasing value of variable k as the innermost loop is executed.

An appropriate index transformation matrix T is selected to achieve the above goal for the restructured array. This is easiest to explain in terms of the algebraic representation of access pattern. As discussed earlier (see (2.3) on page 18), the transformed index vector y for the restructured array can be expressed directly as an affine function of the iteration vector i:

$$y = (TA)\,i + (Ta) \tag{3.1}$$

where T is the index transformation matrix, A is the (original) access matrix, and a is a constant offset vector. Array restructuring in effect transforms the access matrix A to TA. Thus, given the original access matrix A, we have to choose a transformation matrix T such that TA has the nonzero structure we want: the last column of the matrix starts with a zero. Note that there are many transformation matrices that can achieve this. All of them are "equally good" in this respect, although we will discuss later why some are preferred over others.

The transformation matrix given in Figure 3.1 produces the desired effect on the access matrix, and thus on the geometrical representation and the code as well. Geometrically, the corresponding linear transformation maps the original, diagonal vector at the top to the horizontal vector below. The transformed access is generated from the transformed access matrix *TA*. It goes through elements of the restructured array X2 row by row, as desired.

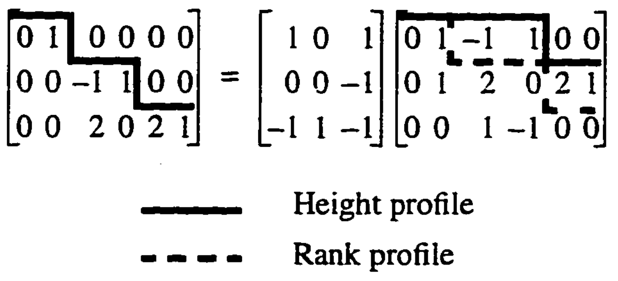
Having discussed the two-dimensional case, we now turn to the general case where the array may have more dimensions. The goal remains the same: the innermost loop accesses elements consecutive in memory. In algebraic terms, this requires that the access matrix's last column, which corresponds to the innermost loop, contains one or more zeros followed by a single nonzero. (Strictly speaking this nonzero must be 1 or -1 for accesses to be consecutive. See page 45 for more discussion on the extent to which we can affect the magnitudes of memory strides.) Given the original access matrix *A*, we choose a transformation matrix *T* such that *TA* satisfies this condition. We say that *T flattens* the column because its effect on the column is to lower the latter's *height*, which is defined as the position of the top nonzero counted from the bottom.

In summary, to improve the spatial locality of an array access by array restructuring, we compute a transformation matrix *T* from the access matrix *A* such that the last column of their product *TA* has one or more leading zeros followed by a single nonzero value or, in terms of column height, has a height of 1. So far, we have focused entirely on the innermost loop, which is undoubtedly most important for locality. The next section considers the other, enclosing loops as well.

3.1.2 Enclosing Loops

So far we have focused on the last column of the access matrix because it corresponds to the innermost loop, which is most important for locality. Specifically, we look for a transformation matrix to flatten this column to a height of one. For the very same reason, we also want to flatten the other columns. In this section, we discuss the inherent limit to how far columns can be flattened and its implications.

$$B = TA$$



$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & -1 \\ -1 & 1 & -1 \end{bmatrix} \begin{bmatrix} 0 & 1 & -1 & 1 & 0 & 0 \\ 0 & 1 & 2 & 0 & 2 & 1 \\ 0 & 0 & 1 & -1 & 0 & 0 \end{bmatrix}$$

Figure 3.2: Flattening Columns of an Access Matrix

An example suffices to show that it is not possible to flatten all the columns to arbitrary heights using the same *nonsingular* transformation matrix. Consider the fourth and sixth columns of the access matrix A and transformed access matrix B in Figure 3.2. Let A_k and B_k be the k-th columns of A and B respectively. For the sake of argument, assume that for some nonsingular transformation matrix, say S, both B_4 and B_6 have a height of one. In other words, both columns have a single nonzero, and it is the last element in each case. Then, $B_4 = \gamma B_6$ for some nonzero scalar γ. Since the transformation matrix S is nonsingular, we can express the columns of A in terms of the corresponding columns of B. Thus,

$$\begin{aligned} A_4 &= S^{-1} B_4 = S^{-1}(\gamma B_6) = \gamma S^{-1} S A_6 \\ &= \gamma A_6 \end{aligned} \tag{3.2}$$

However, this is clearly impossible because both the first and last elements of A_6 are zero whereas those of A_4 are nonzero. Hence, there is no nonsingular transformation matrix that can flatten both the fourth and sixth columns of A to a height of one.

If we cannot flatten columns to arbitrary heights with nonsingular transformation matrices, how far can we go? Before answering this question, we have to define some terms for the following discussion. Let the *height* of a matrix be the maximum of the heights of its columns. For an $m \times n$ access matrix A, let h_j $(1 \leq j \leq n)$ be the height of the submatrix comprising columns j through n. We call the sequence of h_j the *height profile* of the access matrix A. Similarly, let r_j be the rank[4] of the same submatrix and call this sequence the *rank profile* of A. Figure 3.2 illustrates these two profiles. Both the height and rank profiles increase or remain unchanged with decreasing j; in other words, they rise (strictly speaking, do not fall) as we go through the matrix columns from right to left.

The access matrix's rank profile inherently limits how far its columns can be flattened by a nonsingular transformation matrix. Specifically, given the access matrix A, the columns of TA can be flattened at most to match the rank profile of A, as illustrated in Figure 3.2. In other words, the best (i.e., lowest) achievable height profile of TA coincides with the rank profile of A. This assertion follows from two observations.

- First, TA has the same rank profile as A, for the following reason. Since T is nonsingular, left-multiplying it to any matrix preserves the rank of the latter. Therefore, all corresponding submatrices of A and TA have the same rank. In particular, this statement applies to submatrices that consist of columns j through n for any j between 1 and n. Since the rank profile is defined by the ranks of these submatrices, A and TA have the same rank profile.

- Second, for any matrix, the height profile must always be "on or above" the rank profile. More precisely, $h_j \geq r_j$ for any j between 1 and n. In Figure 3.2, the two profiles coincide in the case of matrix B on the left, whereas for matrix A the height profile is "above" the rank profile at some columns.

  To see why this is true in general, let us assume for the sake of argument that $h_j < r_j$ for some j. Consider the columns j through n. By the definition of the height profile, these columns have heights of at most h_j. Therefore, in each of these columns, at most the bottom h_j elements are nonzero; the other elements must be zero. Thus, truncating the leading zero elements, we get a number of h_j-dimensional vectors.

4. The *rank* of a matrix is the number of linearly independent rows it has, which always equals the number of linearly independent columns [Bloom 1979].

On the other hand, by the definition of the rank profile, we know that the submatrix consisting of columns j through n has rank r_j. In other words, there are r_j linearly independent columns among these columns, and thus also among their truncated, h_j-dimensional counterparts. (Recall that all the truncated components are zeros.)

However, this leads to a contradiction if $h_j < r_j$ because in any h_j-dimensional vector space, it is impossible to find more than h_j linearly independent vectors. Hence, $h_j \geq r_j$ for any j between 1 and n. In other words, the height profile must be "on or above" the rank profile.

From these two observations, we see that multiplying a nonsingular transformation matrix T to A may change the height profile but never the rank profile of A. By flattening columns, we can "lower" the height profile at most to the extent that it coincides with the (unchanged) rank profile. Note, however, that so far we have said nothing about how one might find such a transformation matrix, or even whether such a matrix exists at all for a given access matrix. These are discussed later in Section 3.2.2.

A matrix whose height profile coincides with its rank profile would be, in a loose sense, lower-triangular. An example is the matrix B in Figure 3.2. By "lower-triangular," we do not mean "lower-triangular" in the strict mathematical sense; in fact, the matrix in question may not even be square. Instead, we mean that the matrix elements lying above the height profile (which, being identical to the rank profile, rises at most one row per column as we go from right to left through the matrix) constitute an "upper triangle" filled with zeros.

In summary, in the case of a single array access, optimizing for spatial locality requires solving the following mathematical problem: given the original access matrix A, compute a transformation matrix T such that TA has the lower-triangular form described above. We present an algorithm for this purpose later in this chapter.

3.1.3 Unimodular Index Transformation Matrix

In the previous section, we have discussed one requirement on the transformation matrix *T*: it should transform the access matrix *A* such that *TA* has a certain nonzero pattern. We now consider a requirement on the matrix *T* itself. How to find a transformation matrix satisfying both requirements is discussed shortly.

In a word, we require that *T* be unimodular. (This requirement can be relaxed if we restructure only part of an array. Chapter 5 elaborates on this.) A unimodular matrix is an integral, square matrix whose determinant is either 1 or -1. A matrix is unimodular if and only if both the matrix itself and its inverse are integral [Schrijver 1986]. This property is important for our purpose for two reasons.

First, the transformation matrix *T* itself should be integral. Since only integers can be valid array indices, we need to ensure that any integral vector (the original array index vector) is transformed to an integral vector (the transformed array index vector). An integral transformation matrix *T* is clearly a sufficient condition. It is also a necessary condition because the *j*-th column of *T* is the image of an integral vector — the vector with a one in the *j*-th position and zeros elsewhere — and therefore each column of *T* must be integral.

Second, in order that memory is used efficiently, the inverse of the transformation matrix, T^{-1}, must also be integral. Consider the example in Figure 3.3. Although *T* maps every integral vector to an integral vector, not every integral vector in the transformed index space is the image of some integral vector in the original index space. Thus, some elements of the restructured array (such as element `[1,1]`) do not correspond to elements of the original array and therefore do not contain array data — even though they do take up memory. If T^{-1} is integral, however, every integral vector would be an image of some integral vector under the transformation *T* (the latter vector obtained by multiplying T^{-1} to the former). Therefore, all elements of the restructured array are used. (For the moment, we ignore the array bounds. Issues related to the transformation of array bounds are dis-

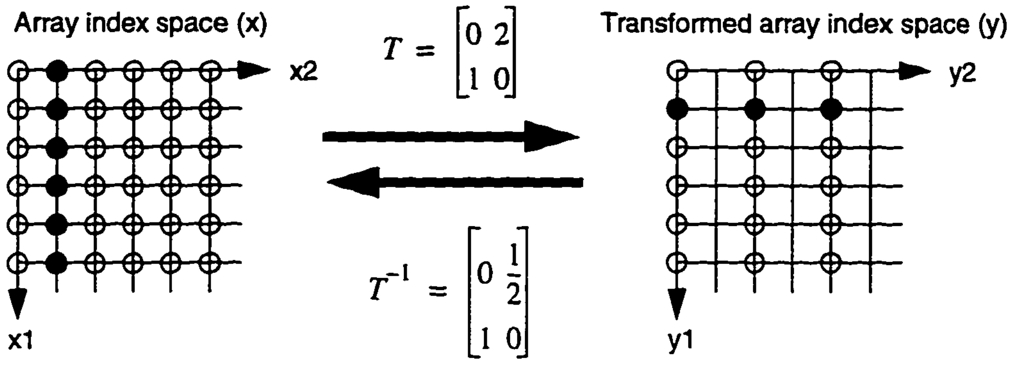

Array index space (x)
x2
x1
$T = \begin{bmatrix} 0 & 2 \\ 1 & 0 \end{bmatrix}$
$T^{-1} = \begin{bmatrix} 0 & \frac{1}{2} \\ 1 & 0 \end{bmatrix}$
Transformed array index space (y)
y2
y1

Figure 3.3: Effect of Transformation Matrix with Non-Integral Inverse

cussed in Chapter 4.) In short, an integral T^{-1} is a sufficient condition for using all restructured array elements. Moreover, it is also a necessary condition. The argument examines each column of T^{-1} and resembles the earlier argument on the necessity of an integral T.

3.1.4 Summary

To sum up, given the access matrix A for an array access, we wish to find an index transformation matrix T such that T is unimodular and TA has the "lower-triangular" form discussed earlier — with zeros in the "upper triangle" that lies above the rank profile of A. The next section discusses how to compute such a transformation matrix.

3.2 Computing the Index Transformation Matrix

We now explain how to compute an index transformation matrix T satisfying the above requirements. For clarity in exposition, we start with a simple algorithm that considers only a special class of unimodular matrices useful in many common cases. This is followed by a more general algorithm that does not have such *a priori* restrictions.

3.2.1 Permuting Array Indices

In this section, we describe a simple algorithm to compute an index transformation matrix T from the access matrix A. Essentially, it selects only matrices that represent permutations of array indices. Though more restrictive than the general algorithm presented later, it is useful in some simple cases and follows the same overall solution approach. Therefore, discussion here also helps to lay the groundwork for subsequent discussion of the general algorithm.

The algorithm is restrictive in that it considers only permutation matrices as candidates for the index transformation matrix. A permutation matrix is a square matrix in which each row and each column contains a single nonzero, which equals 1. Left-multiplying a permutation matrix to a (column) vector, as we do when applying an index transformation, produces a vector whose components are some permutation of the original vector's components[5]. Thus, with a permutation matrix, we in effect restructure an array by permuting the array indices and storing the elements in row-major order according to the permuted indices. For example, for the permutation matrix $\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$, element `[x1,x2,x3]` of a three-dimensional original array corresponds to element `[x2,x1,x3]` of the restructured array. Two special cases are row-major storage (the transformation matrix is the identity matrix) and column-major storage (the transformation matrix has ones along the off-diagonal, reversing the order of the array indices). For two-dimensional arrays, these are the only possible permutations. In general, for an m-dimensional array, there are $m!$ possible permutations.

Considering only permutation matrices simplifies the task of finding T. Any permutation matrix is unimodular because it is integral (each element is either 0 or 1) and its determinant is 1 or -1. Therefore, our problem reduces to finding a permutation matrix T such that TA has the desired nonzero structure.

5. Similarly, right-multiplying a permutation to a row vector also permutes the vector components. Since we work with column vectors only, this property does not concern us.

```
m = number of array dimensions = number of rows in A
n = number of loop levels number of columns in A

A = access matrix                     A is an m by n array
T = m by m identity matrix            T is an m by m array
j = m                                 j indexes a row
k = n                                 k indexes a column

WHILE (j > 1) and (k >= 1) DO
  Permute rows 1 through j to bring zeros in A[1..j,k] to the top
  Permute rows 1 through j of T accordingly
  j = row position of last zero element in A[1..j,k] or 0 if none
  k = k - 1
END
```

Figure 3.4: Algorithm to Transform Access Matrix with Permutation Matrix

The algorithm for computing *T* from *A* hinges on the fact that multiplying a permutation matrix *T* to *A* in effect permutes the rows of *A*. Intuitively, since we permute the array indices, we also permute the corresponding rows of the original access matrix accordingly to obtain the transformed access matrix.

The algorithm permutes the rows of *A* to produce the desired nonzero structure; the permutation matrix *T* is obtained by applying the same permutation to the identity matrix. Figure 3.4 shows the algorithm. Figure 3.5 illustrates its operation on an example. The two columns of matrices show how the arrays `A` (the access matrix being transformed) and `T` (the transformation matrix being computed) change as the algorithm permutes rows.

At the beginning, when `j` equals `m` and `k` equals `n`, the algorithm focuses on the last column of the entire matrix. The algorithm permutes the rows to bring the zeros to the top of the column, thus flattening the column (i.e., lowering the position of the top nonzero as far as possible). The same permutation is applied to the transformation matrix being formed. In the special case where the entire column is zero, no action is required to flatten the column: the column already has a height of zero. Permuting the rows is valid, though superfluous. On the other hand, if the entire column is nonzero, nothing can be done: row permutation can move the nonzeros, but not eliminate them.

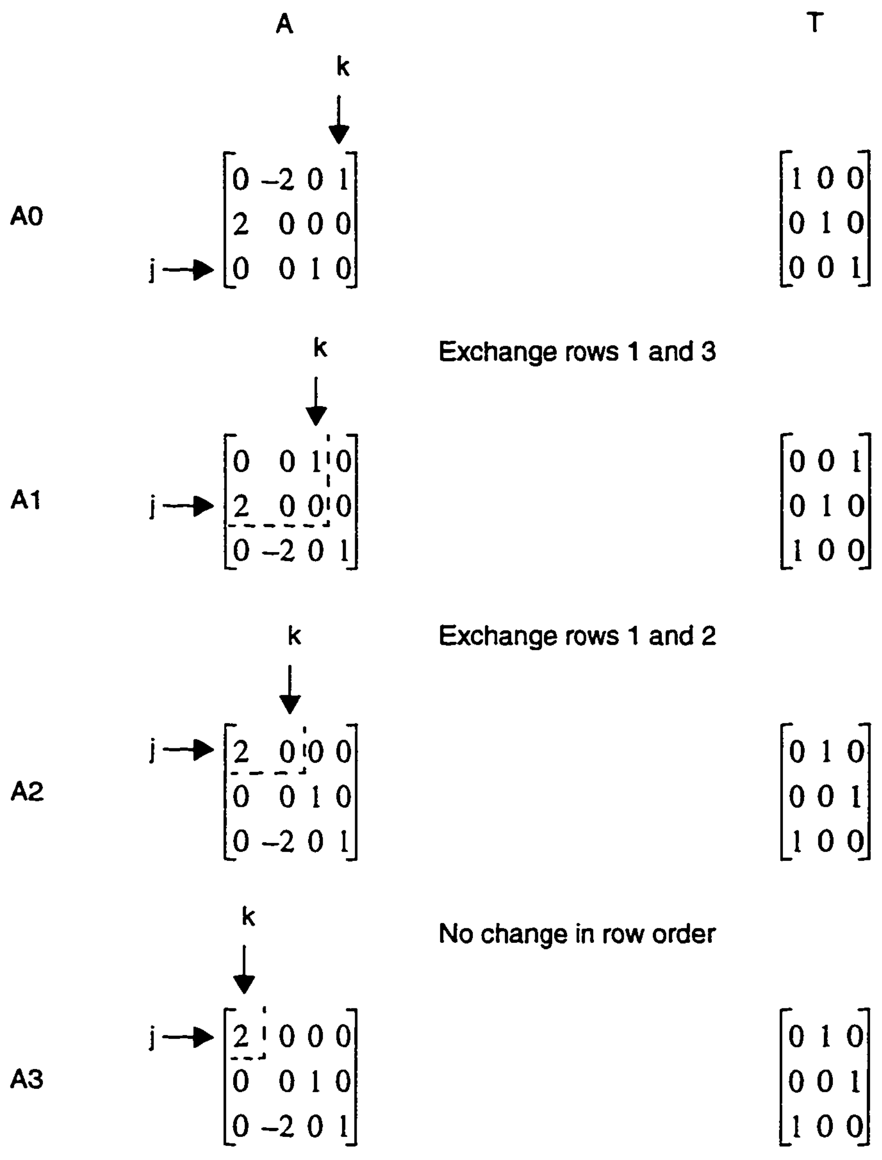


Figure 3.5: Transforming Access Matrix by Permuting Rows

After flattening the last column, the algorithm shifts its attention to the submatrix consisting of the remaining columns and the rows that contain zeros in the last column. (This submatrix is delimited by variables j and k in the algorithm and by the dashed line in Figure 3.5.) Future permutation of these rows will not affect the columns we have processed because the matrix elements in the intersection of these rows and columns are zero anyway. At each step, the submatrix shrinks by one column and possibly one or more rows. The algorithm terminates when the submatrix shrinks into a single row or vanishes altogether, at which point further row permutations become meaningless.

Notice that we have not specified exactly how rows should be permuted to bring zeros to the top of a column. There may be multiple ways of doing this, but they lead to results that are equivalent in having the same height profile. For instance, consider the example in Figure 3.5. In the first step, we could have brought the two zeros in the last column to the top by moving the second and third rows up and the first row to the bottom. If we had done so, we would have obtained A2 directly without going through the second step. In the end, the algorithm would have computed the same transformation matrix. In general, no matter how we permute rows to bring zeros in a column to the top, the number of leading zeros in the resulting column is the same. Therefore, so is the height of that column and consequently the eventual height profile of the matrix.

This simple algorithm works best when each column contains at most one nonzero, which means that each loop variable appears in at most one array index (but an array index may involve more than one loop variable). Figure 3.5 is one example. Intuitively, we consider the loops one by one starting from the innermost. If the loop variable appears in an array index (there is at most one), we move this array index to the next less rapidly varying index position.

However, if some column contains multiple nonzeros (i.e., some loop variable appears in multiple array indices), permuting array indices may not be enough to make the innermost loop go through elements consecutively. One example is the array access we considered in Section 3.1.1 (shown in bold in Figure 3.1 on page 26). Since the array has only two dimensions, there are only two possible permutations: no transformation or array transpose. As we discussed before, array transpose does not solve the problem because with or without transpose, the innermost loop accesses array elements in different rows *and* different columns. For this and similar cases, we must use general unimodular matrices. The following section discusses an algorithm to choose one.

3.2.2 General Unimodular Transformation Matrices

Given the simple introduction afforded by our consideration of only permutation matrices, we now present the general algorithm for computing a unimodular index transformation matrix from the access matrix. This algorithm always finds a transformation that flattens the access matrix such that the resulting height profile coincides with the rank profile. As discussed earlier in Section 3.1.2, no nonsingular transformation matrix can flatten the access matrix any further.

The algorithm consists of two steps. First, it finds a nonsingular matrix S such that the height profile of SA coincides with the rank profile of A. However, S may or may not be unimodular. In fact, it is not even guaranteed to be integral. Therefore, the second step computes from S a unimodular matrix T such that TA has the same height profile as SA. In this sense, T is "as good as" S, but in addition it satisfies the unimodularity requirement. In the rest of this section, we describe the algorithm to find S, show that the height profile of SA must be the same as the rank profile of A, and finally discuss the algorithm to find T from S.

Finding Nonsingular Transformation Matrix S

The first step resembles the simple algorithm discussed earlier in Section 3.2.1, but with one crucial difference: instead of only permuting rows, we use elementary row operations. In short, the algorithm transforms the original access matrix A to the desired form by means of elementary row operations, and accumulates the effects of these operations in a nonsingular matrix computed by applying the same sequence of operations to the identity matrix.

```
m = number of array dimensions = number of rows in access matrix
n = number of loop levels = number of columns in access matrix

A = access matrix                  A is an m by n array
S = m by m identity matrix         S is an m by m array
j = m                              j indexes a row

FOR k = n DOWNTO 1 DO              k indexes a column

  IF (A[1..j,k] are not zero) THEN

    Ensure that A[j,k] is nonzero
    IF (A[j,k] = 0) THEN
      Find a nonzero in A[1..(j-1),k]. Suppose it is in row i.
      Exchange row i of A and row j of A
      Exchange row i of S and row j of S
    ENDIF

    Make A[1..(j-1),k] zero by eliminating nonzeros
    FOR i = 1 TO j-1 DO
      f = A[i,k]/A[j,k]
      Multiply row j of A by f and subtract from row i of A
      Multiply row j of S by f and subtract from row i of S
    ENDFOR

    Update only rows 1 through j-1 in future
    j = j - 1

  ENDIF

ENDFOR
```

Figure 3.6: Algorithm to Transform Access Matrix with Nonsingular Matrix

There are three types of elementary row operations: scaling a row by a nonzero factor, exchanging two rows, and adding a multiple of one row to another [Schrijver 1986]. They subsume row permutations: any row permutation is equivalent to a series of row exchanges. As noted in Section 3.2.1, row permutation can only move nonzeros, but not eliminate them. The multiply-and-add operation can eliminate nonzeros.

The algorithm is shown in detail in Figure 3.6. It resembles the Gaussian elimination algorithm for inverting a matrix but applies to square as well as non-square matrices. All arithmetic operations are performed on rational numbers, rather than integers or floating point numbers. The algorithm has $O(mn^2)$ complexity for an $m \times n$ access matrix, where m is the number of array dimensions and n is the number of loop levels. This is because the k-loop is executed n times, the i-loop at most m times, and each elementary row operation takes n steps, one for each element in the row. The cost of the algorithm is small because there are typically only a handful of array dimensions and loop levels.

The algorithm starts from the last column (when `k` is `n`). If the entire column is zero, no action is needed: the height is already as small as it can possibly be. If the column contains nonzeros, we ensure that the bottom element is nonzero by a row exchange if necessary. (This is similar to pivoting in Gaussian elimination.) Then, we eliminate all other nonzeros, if any, by multiply-and-add row operations, thus obtaining a column with only one nonzero in the bottom row and therefore a height of 1. The same sequence of row operations are applied to the identity matrix to compute the corresponding transformation matrix in `S`.

After flattening the last column, the algorithm drops the last row and column from further consideration and focuses on the remaining submatrix (delimited by variables `j` and `k`). Future updates on the remaining rows (i.e., rows 1 through `j`) will not affect the columns that have been processed and dropped from consideration (i.e., columns `k+1` through `n`) because the intersection of these rows and columns contains only zeros. The algorithm terminates after going through all columns. (In fact, it can terminate earlier: when `j` reaches 1, the loop body in the algorithm performs no more updates to `A` and `S`.)

Guaranteeing Identical Height and Rank Profiles

The above algorithm always finds a transformation matrix S such that the height and rank profiles of SA coincide. As noted in Section 3.1.2, no nonsingular transformation matrix can flatten the access matrix A any further. We now explain why it is guaranteed to achieve this.

The key is to show that after iteration k — the iteration for which the loop variable k in Figure 3.6 has the value k, not the k-th iteration — the submatrix comprising columns k through n of matrix variable A (i.e., the columns flattened so far) is of height $m - j_k$ as well as rank $m - j_k$, where j_k is the value of variable j at the end of iteration k (after the decrement, if any, in iteration k). To simplify the following argument, we augment A with a column $n + 1$ that contains only zeros and our algorithm with an iteration $n + 1$ that does nothing. Adding a zero column to any matrix has no effect on the matrix's rank and height. Therefore, the extra column does not affect the rank and height profiles. For the augmented A, we claim that

$$rank(A_k^{(l)}) = height(A_k^{(l)}) = m - j_k \qquad \text{for } 1 \le l \le k \le n + 1 \qquad (3.3)$$

where $A_k^{(l)}$ is the submatrix consisting of columns k through $n + 1$ of matrix variable A at the end of iteration l, while $rank(\ldots)$ and $height(\ldots)$ denote the rank and height of a matrix respectively. (This claim will be justified shortly.) When the algorithm terminates in iteration 1 ($l = 1$), this claim applies to all k between 1 and $n + 1$ (inclusive). Therefore, the height and rank profiles of the final A — which is the transformed access matrix SA augmented with a zero column — are identical: $rank(A_k^{(1)}) = height(A_k^{(1)})$ for $1 \le k \le n + 1$. At the same time, the corresponding transformation matrix S has been computed in the matrix variable S.

We now justify the claim in (3.3) by examining how each iteration updates A. Formally, we prove the claim by induction on l, which decreases from $n + 1$ to 1.

We first examine the initial case: $l = n + 1$. Variable j is initialized to the number of rows. Thus, $j_{n+1} = m$. The submatrix $A_{n+1}^{(n+1)}$ consists of a single zero column (which we have augmented A with) and hence has height and rank 0. Thus, (3.3) holds true.

Now consider the situation at the end of iteration l ($l \le n$), with the assumption that (3.3) holds at the end of the previous iteration (i.e., for $l + 1$).

First, note that (3.3) remains true for $k \geq l+1$ because iteration l does not change the columns processed in previous iterations, namely columns $l+1$ through $n+1$. That is,

$$A_{l+1}^{(l)} = A_{l+1}^{(l+1)} \tag{3.4}$$

To see this, note that at the end of iteration $l+1$ (the previous iteration), our induction hypothesis implies that $height(A_{l+1}^{(l+1)}) = m - j_{l+1}$ (by letting $k = l+1$). Therefore, the top nonzero in $A_{l+1}^{(l+1)}$ is in row $j_{l+1}+1$, while rows 1 through j_{l+1} of $A_{l+1}^{(l+1)}$ are all zero. Since the updates in iteration l (the current iteration) are restricted to these rows of matrix variable A, they do not affect columns $l+1$ through $n+1$.

What remains to be shown is that the claim holds true also for $k = l$, that is, for the submatrix including the just processed column l. In other words, we need to show that

$$rank(A_l^{(l)}) = height(A_l^{(l)}) = m - j_l \tag{3.5}$$

There are two cases to consider: the top j_{l+1} elements of column l are all zero, or they are not.

- In the first case, iteration l does nothing: the variable j is not changed; nor does including the unmodified column l change the rank and height of the submatrix of processed columns. More specifically, $j_l = j_{l+1}$ because j is not modified. Moreover, since iteration l simply expands the submatrix of processed columns by the unmodified column l, the resulting submatrix $A_l^{(l)}$ consists of column l and the columns in $A_{l+1}^{(l+1)}$. Note that $rank(A_{l+1}^{(l+1)}) = height(A_{l+1}^{(l+1)}) = m - j_{l+1}$ by the induction hypothesis, and that column l has j_{l+1} leading zeros in this case. We first determine $rank(A_l^{(l)})$. We observe that $rank(A_l^{(l)}) \geq rank(A_{l+1}^{(l+1)}) = m - j_{l+1}$ because $A_l^{(l)}$ contains all the columns in $A_{l+1}^{(l+1)}$. We also observe that $rank(A_l^{(l)}) \leq m - j_{l+1}$ because every column of $A_l^{(l)}$ (either column l or a column from $A_{l+1}^{(l+1)}$) has nonzeros only in the bottom $m - j_{l+1}$ elements. Combining these two observations, we have $rank(A_l^{(l)}) = m - j_{l+1}$. By an analogous argument, $height(A_l^{(l)}) = m - j_{l+1}$. These two results together give us

$$rank(A_l^{(l)}) = height(A_l^{(l)}) = m - j_{l+1} = m - j_l \qquad (3.6)$$

- In the second case, where some of the top j_{l+1} elements of column l are nonzero, iteration l decrements `j` and updates `A` appropriately. Thus, $j_l = j_{l+1} - 1$. Also, $A_l^{(l)}$ consists of the updated column l and the columns of $A_{l+1}^{(l)}$, which is identical to $A_{l+1}^{(l+1)}$ according to (3.4). Note that column l is updated in such a way that it has $j_{l+1} - 1$ leading zeros followed by a nonzero in row j_{l+1}. Thus, the updated column l must have a height of $m - j_{l+1} + 1$. Note also that by the induction hypothesis $A_{l+1}^{(l+1)}$ has a height of $m - j_{l+1}$. Therefore, $height(A_l^{(l)})$ equals $m - j_{l+1} + 1$, the greater of the two heights. As for $rank(A_l^{(l)})$, $A_l^{(l)}$ contains $m - j_{l+1} + 1$ linearly independent columns: $m - j_{l+1}$ of them from $A_{l+1}^{(l+1)}$ (whose rank is $m - j_{l+1}$) and the last one being the updated column l (which is linearly independent of all columns in $A_{l+1}^{(l+1)}$ because of its unique nonzero in row j_{l+1}). Thus, $rank(A_l^{(l)}) = m - j_{l+1} + 1$. Combining the results for the rank and height, we have in this second case

$$rank(A_l^{(l)}) = height(A_l^{(l)}) = m - j_{l+1} + 1 = m - j_l \qquad (3.7)$$

This completes the proof for the claim in (3.3). It follows that our algorithm finds a transformation matrix S such that SA has identical height and rank profiles.

Finding Unimodular Transformation Matrix T

So far, we have described the first step of our algorithm: finding a nonsingular matrix S such that SA has the desired form. However, S may or may not be unimodular; in fact, it may not even be integral. In the next step, we compute from S a unimodular matrix T such that TA and SA have identical height profiles. In this sense, T and S are "equally good."

In a word, we choose $T = H_S^{-1} S$, where H_S is the Hermite normal form of S. A rational matrix is in Hermite normal form if and only if (a) it is lower-triangular, (b) all its elements are nonnegative, (c) the diagonal element in each row is greater than other elements in the same row [Schrijver 1986]. It is known that for any rational nonsingular matrix, say B, there exists a unimodular matrix U such that $H = BU$ is in Hermite normal form. Moreover, for any such matrix B, the corresponding Hermite normal form matrix is unique. Thus, it is meaningful to talk about *the* Hermite normal form of a matrix. There are known algorithms to compute it [Schrijver 1986].

To justify our choice for T, we must first show that H_S^{-1} does exist. Then, we show that $T = H_S^{-1} S$ satisfies both our requirements: T is unimodular, and TA has the same height profile as SA.

- H_S^{-1} exists because $|H_S|$ is nonzero: the determinant of H_S is the product of its main diagonal elements (because it is lower-triangular); and those elements are all positive (because each of them is greater than other elements on the same row, which are nonnegative).

- To see that T is unimodular, note that

$$T = H_S^{-1} S = (SU)^{-1} S = U^{-1} S^{-1} S = U^{-1} \tag{3.8}$$

  for some unimodular matrix U. Since the inverse of a unimodular matrix is also unimodular [Schrijver 1986], T is unimodular.

- As for the height profile requirement, note that H_S is lower-triangular, by the definition of Hermite normal form. Therefore, so is its inverse H_S^{-1}. Moreover,

$$TA = \left(H_S^{-1} S\right)A = H_S^{-1} (SA) \tag{3.9}$$

  In other words, we can obtain TA by left-multiplying SA with a lower-triangular matrix, namely H_S^{-1}. This does not change the heights of the columns, and therefore TA and SA have the same height profile.

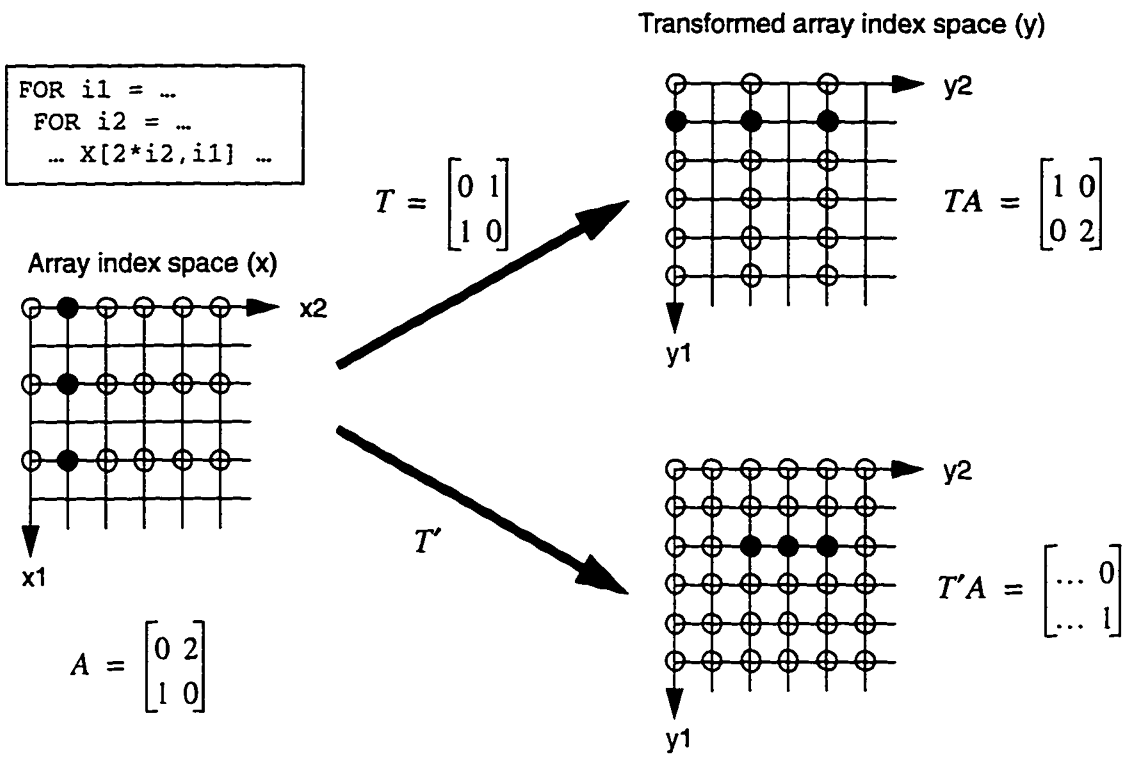


Figure 3.7: Affecting Magnitudes of Strides in Index Space

Magnitudes of Strides in Index Space

So far, we have focused on the directions of strides in the index space. Could we also reduce the magnitudes of the strides to improve spatial locality? Unfortunately, we cannot reduce these magnitudes at will as long as the restructured array must contain all elements of the original. Consider the example in Figure 3.7. The upper half shows the computed index transformation. With this transformation, the innermost loop accesses elements in the same row as indicated by the row of three black circles.

One might want to reduce the innermost loop's stride (i.e., the top, and in fact only, nonzero in the last column of TA) from 2 to 1. The lower half of the diagram shows this hypothetical situation. The black elements, accessed by consecutive iterations, are adjacent, indicating a unit stride in the innermost loop. However, no linear transformation can achieve this because with any linear transformation, the elements between the black elements in the original index space remain between the black elements in the transformed index space. (This is inherent to linear transformations: they map colinear points to colinear points.) There is no "space" between the adjacent black elements in the restructured array. Nevertheless, we can have some control over the magnitudes of strides by omitting unaccessed elements (such as those between the black ones) from the restructured array completely. Chapter 5 discusses this in detail.

Summary

Given an access matrix A, we compute an index transformation matrix T that improves locality of the access in two steps.

1. Use the algorithm in Figure 3.6 on page 39 to compute from access matrix A a nonsingular matrix S such that SA has the desired form. SA is guaranteed to have identical height and rank profiles. No nonsingular transformation matrix can flatten the access matrix further than that.

2. Let $T = H_S^{-1}S$, where H_S is the Hermite normal form of S, which can be computed using known algorithms. The final index transformation matrix T is guaranteed to be unimodular. Moreover, TA has the same height profile as SA, and in this sense T is just "as good as" S for our purpose of improving locality.

3.3 More General Access Patterns

We have fully discussed a simple, but restrictive, case: a single access with affine index expressions to the array in question in a single perfect loop nest. In any real application, many of these simplifying assumptions do not hold. In this section, we discuss how to handle more general cases:

- multiple accesses to the same array
- non-affine array index expressions
- imperfect loop nests
- multiple loop nests

3.3.1 Multiple Accesses

So far, we have focused on just one array access and discussed how to compute an index transformation matrix from one access matrix. In general, however, a loop nest may contain multiple accesses to the same array. In fact, the example loop nest in Figure 3.1 of Section 3.1.1 contains two accesses to each of the two read-only arrays (X and Y), even though we concentrated on only one access in our earlier discussion.

In this section, we discuss the handling of multiple accesses to the same array. We first explore two broad issues: whether or not to create multiple, differently transformed copies of the array for different accesses, and exploiting locality between accesses in the same iteration. Then, we describe the more specific question of how we compute our index transformation matrix from the access matrices for multiple accesses.

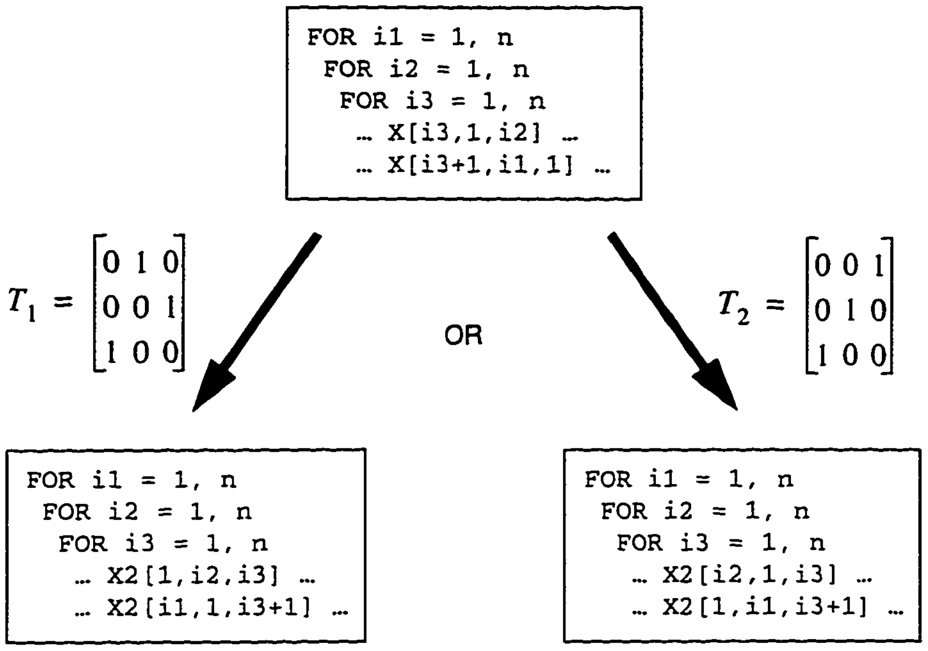
Multiple Copies or One Copy

Sometimes, there is simply no one array layout that yields good locality for all accesses. A simple example is a two-dimensional array accessed by the same loop in row-major order through one access and in column-major order through another. One of the accesses is bound to exhibit poor locality whether we store the array in row-major or column-major order. Even if we allow more general layouts, as our framework does, at least one of the accesses cannot go through memory consecutively as desired. In fact, in this case, other layouts only worsen the situation by causing both accesses to have bad locality.

This problem can be addressed by creating multiple versions of the same array, each laid out differently. At one extreme, each array access has its own version. Alternatively, accesses that are "similar enough" to one another may share a single version. However, this approach has several drawbacks. First, the array has to be read-only. If the array may be written, we would have to maintain consistency between the multiple versions by updating all copies of the same element whenever one is updated (analogous to maintaining cache coherence on multiprocessors, but with no hardware support). This is both cumbersome and expensive. Second, the amount of memory required for the restructured array is increased by the duplication factor. So is, roughly, the runtime copying overhead for creating the restructured array. Third, since the loop accesses these array versions simultaneously, the competition for limited cache space is intensified. This may lead to more cache misses and thus diminish, if not outweigh, the performance advantage gained by improving the locality of each individual array access. Similarly, given a fixed amount of memory, the largest problem size that can be handled without excessive paging is reduced.

In many cases, however, one array layout suffices for a number of similar, but not identical, array accesses. One common case is the case of uniformly generated accesses [Wolf and Lam 1991a]. Uniformly generated accesses are accesses that vary with the loop variables in identical ways and differ only in the constant offsets, such as `X[i+j-1,j]` and `X[i+j,j-2]`. In formal terms as defined in (2.1) on page 15, they have the same access matrix A, but different offset vectors a. Since our algorithm for computing the transformation matrix depends only on the access matrix, such accesses would yield exactly the same transformation matrix and thus can share the same array layout. Intuitively, different uniformly generated accesses move through the array index space in the same directions when execution goes through various loops; they differ only in their starting points. Since we are concerned with only these directions of movement, we can treat a set of uniformly generated accesses as a single access.

Even if a number of array accesses are not uniformly generated, they may still be similar enough such that a single array layout is "good enough," though not ideal, for all of



```
FOR i1 = 1, n
 FOR i2 = 1, n
  FOR i3 = 1, n
   … X[i3,1,i2] …
   … X[i3+1,i1,1] …
```

$T_1 = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$ OR $T_2 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$

```
FOR i1 = 1, n
 FOR i2 = 1, n
  FOR i3 = 1, n
   … X2[1,i2,i3] …
   … X2[i1,1,i3+1] …
```

```
FOR i1 = 1, n
 FOR i2 = 1, n
  FOR i3 = 1, n
   … X2[i2,1,i3] …
   … X2[1,i1,i3+1] …
```

Figure 3.8: Multiple Accesses to One Array

them. Consider the example in Figure 3.8. The triple loop has two accesses to the three-dimensional array X. Although they are not uniformly generated, they are similar in that only the first array index changes with the innermost loop variable i3. Thus, both transformations in the figure would improve performance substantially because they both allow the innermost loop to go through memory consecutively, even though in each case the array layout is not ideal for one of the accesses (the second for T_1, the first for T_2).

Locality between Accesses in the Same Iteration

In optimizing for locality, we treat each access individually: we focus on the spatial locality between occurrences of one access in consecutive iterations rather than that of multiple accesses to the same array in one iteration. Consider the illustrative example in Figure 3.9. If array X is stored in row-major order, as it is originally, the four accesses would touch adjacent array elements in any single iteration, whereas consecutive iterations would touch elements in different rows. If X is transposed, accesses in the same iter-

```
FOR i = 1, b, 4
 FOR j = 1, b
  … X[j,i] …
  … X[j,i+1] …
  … X[j,i+2] …
  … X[j,i+3] …
```

```
FOR i = 1, b, 4
 FOR j = 1, b
  … X2[i,j] …
  … X2[i+1,j] …
  … X2[i+2,j] …
  … X2[i+3,j] …
```

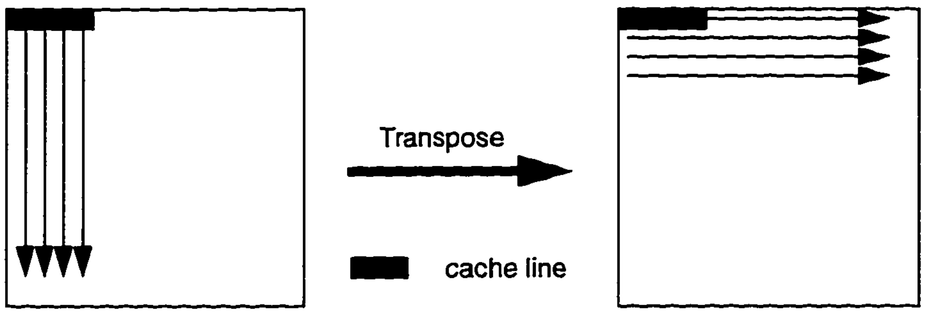


Figure 3.9: Optimizing Multiple Accesses Individually or Collectively

ation would touch elements in different rows, whereas for each access, consecutive iterations touch consecutive elements. These two scenarios are depicted in Figure 3.9. Between these two alternatives, our algorithm would choose to transpose the array because it considers each access individually and therefore does not recognize that the four seemingly unrelated accesses to X in fact touch adjacent memory locations.

The impact of not recognizing and exploiting this is compensated for (and sometimes more than compensated for) by the improved locality of each individual access. After array restructuring, the four transformed accesses lead to four separate streams of consecutive memory references through four distinct memory regions. Occurrences of any one access in consecutive iterations can probably reuse array elements in the same cache line despite intervening accesses to other cache lines because such intervening accesses are few (three to X2 in the example). More generally, suppose the stride were s, instead of 4, each cache line contains l elements, and the value of variable b is b. The original loop would incur roughly $\left\lceil \frac{s}{l} \right\rceil b \left\lceil \frac{b}{s} \right\rceil$ cache misses: $\left\lceil \frac{s}{l} \right\rceil$ per iteration for b iterations in the innermost loop. If the array were transposed, the approximate number of misses would be $\left\lceil \frac{b}{l} \right\rceil s \left\lceil \frac{b}{s} \right\rceil : \left\lceil \frac{b}{l} \right\rceil$ for each of the *s* accesses in one execution of the innermost loop. The latter count is likely to be smaller since *b* is typically much greater than *l*, and thus $\left\lceil \frac{b}{l} \right\rceil s \left\lceil \frac{b}{s} \right\rceil \cong \frac{bs}{l} \left\lceil \frac{b}{s} \right\rceil \le \left\lceil \frac{s}{l} \right\rceil b \left\lceil \frac{b}{s} \right\rceil$. However, for a cache with low associativity, performance may suffer severely if the memory locations accessed in an iteration happen to map to the same position in the cache (typically because they are apart by some power of two). This problem can be remedied by array padding — adding a small, fixed number of dummy array elements to each row to avoid such an unfortunate mapping of cache lines [Bacon et al. 1994].

Computing an Index Transformation Matrix

Based on the general discussion above, we now describe how our compiler handles multiple accesses to the same array. We choose not to use multiple copies of an array in a single loop nest, for the reasons discussed above. Instead, we compute a single transformation matrix from an *aggregate access matrix*, which combines the individual access matrices in a way to be described. The offset vectors are ignored. As we have discussed, they have little effect on locality. By ignoring them, we lose no essential information and automatically treat a set of uniformly generated accesses as one.

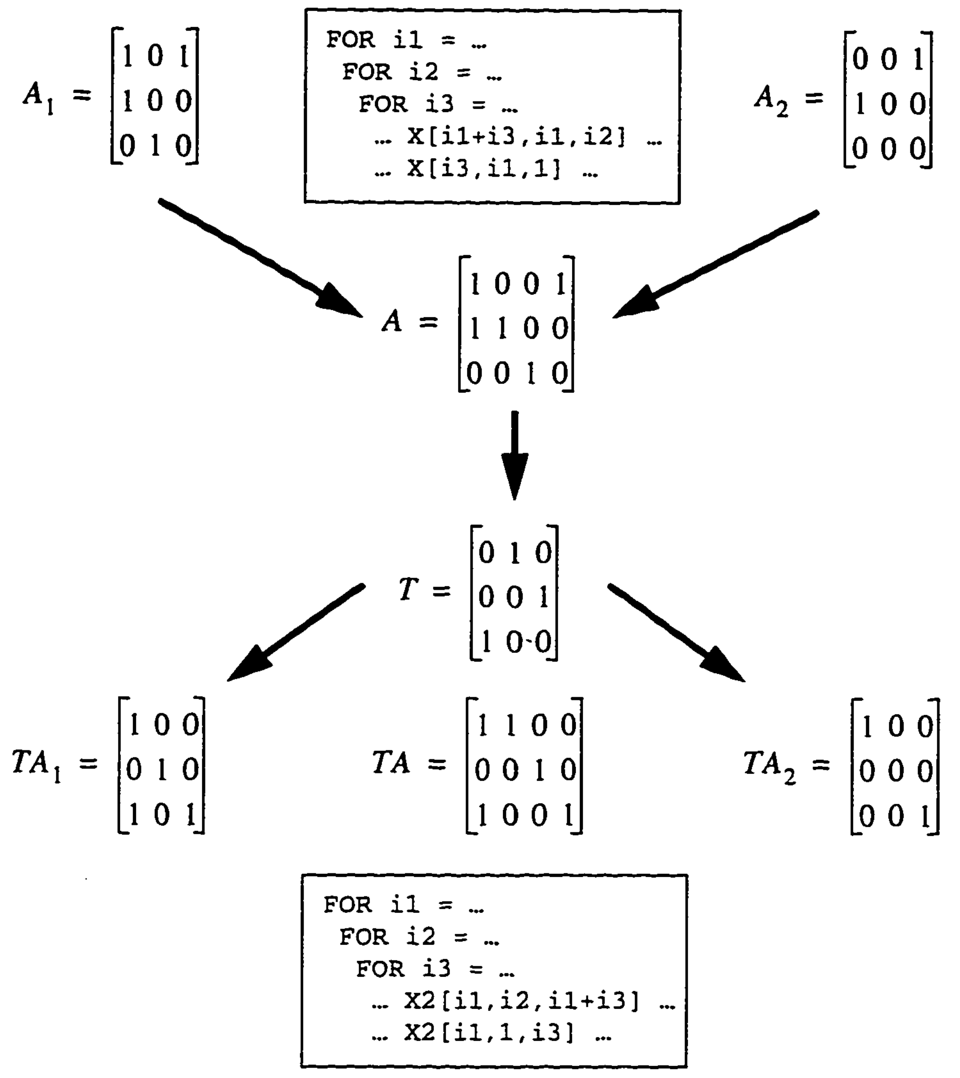
The aggregate access matrix consists of columns from the individual access matrices ordered in a way that reflects their relative importance to locality similar to the single access case. Recall that each column indicates how the array indices change as the corresponding loop variable is incremented. For a single access, we aim to flatten these columns, most of all the one corresponding to the innermost loop because it is most important for locality, but to a lesser extent other columns as well. Since this column is the rightmost in the access matrix, our algorithm transforms the matrix into, loosely speaking, a lower-triangular form for this purpose.

This notion is generalized to multiple accesses. Columns from individual access matrices are ordered in the aggregate matrix according to their corresponding loop levels: the column corresponding to an inner loop is placed to the right of that corresponding to an outer loop; tie-breaking rules for columns corresponding to the same loop level are designed to preserve the original layout as much as possible[6]. Thus, when there is only one access, the aggregate access matrix reduces to the single access matrix.) Figure 3.10 illustrates this. The identical rightmost columns of A_1 and A_2 become the rightmost column of the aggregate matrix A; the middle of A_1 becomes the next in A; the middle of A_2 is omitted as explained below; the leftmost columns of A_1 and A_2 make up the rest of A.

When combining individual access matrices into an aggregate, we can treat certain columns specially. First, zero columns are ignored (as in Figure 3.10). This is because any transformation matrix T would transform a zero column to a zero column. Hence, zero columns do not affect our choice and therefore are dropped altogether. Second, identical columns (e.g., $\begin{bmatrix}1\\0\\0\end{bmatrix}$) and columns differing by only a constant scaling factor (e.g., $\begin{bmatrix}1\\2\end{bmatrix}$ and $\begin{bmatrix}2\\4\end{bmatrix}$) are treated as one because any transformation matrix that flattens one column to a certain height always flattens the other to the same height.

Thus, the algorithm transforming an individual access matrix to lower-triangular form achieves roughly the same effect with the aggregate access matrix: columns most important for locality are flattened most. Figure 3.10 again illustrates this. After choosing the transformation matrix T, we can transform the two accesses individually according to (2.3) on page 18, effectively turning their access matrices A_1 and A_2 into TA_1 and TA_2 respectively. The transformed aggregate access matrix TA has the expected lower-triangular form, but the two individual transformed matrices may not. However, since the rightmost column in both are flattened to a height of one, both accesses now go through memory consecutively, at least in the innermost loop.

6. Specifically, columns are ordered as follows: a column corresponding to an inner loop is placed to the right of one corresponding to an outer loop; that being equal, a column occurring more times (in the individual access matrices) at that level is placed to the right of one occurring fewer times; that being equal, a column with fewer nonzeros is placed to the right of one with more; that being equal, a column with a smaller height is placed to the right of one with a greater height. The last two rules tend to preserve the original array layout, other things being equal.



```
FOR i1 = …
  FOR i2 = …
    FOR i3 = …
      … X[i1+i3,i1,i2] …
      … X[i3,i1,1] …
```

$$A_1 = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \qquad A_2 = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$A = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$T = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

$$TA_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \qquad TA = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \qquad TA_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

```
FOR i1 = …
  FOR i2 = …
    FOR i3 = …
      … X2[i1,i2,i1+i3] …
      … X2[i1,1,i3] …
```

Figure 3.10: Computing Transformation from Aggregate Access Matrix

Summary

In summary, we choose one index transformation for each array, focusing on the potential for reuse between instances of each access in consecutive iterations, rather than that between accesses in the same iteration. To find this transformation matrix, we extract columns from the individual access matrices, order those columns based on their importance to locality, form an aggregate access matrix, and finally apply previously described algorithms to compute the transformation matrix.

3.3.2 Non-Affine Array Index Expressions

We have so far assumed that all array index expressions are affine functions of loop variables, thus allowing analysis and transformation of accesses entirely in a simple linear algebraic framework. To maximize applicability, we now extend our techniques to deal with non-affine index expressions as well. While affine accesses are most common and therefore appropriately the focus of our effort, it would be overly conservative if we were to give up restructuring an array when even a single index expression is not affine.

In dealing with non-affine (or otherwise complicated) accesses, array restructuring promises wider applicability than traditional loop restructuring techniques. Unlike loop restructuring, its legality does not rely on dependence analysis, which may be frustrated by the existence of even a single non-affine index expression in the loop. With array restructuring, affine and non-affine accesses are transformed similarly, as discussed shortly. In other words, non-affine index expressions do not prevent us from applying a selected array transformation. To select the transformation in the first place, we use information from the affine accesses, if any, and whatever can be learnt from the non-affine ones, however limited. At worst, we pick a transformation that does not improve locality as expected, but we never modify the program illegally.

To handle non-affine array accesses, we have to address two issues: mechanism and policy. By mechanism, we mean how to transform array accesses for a given index transformation. We show the transformed accesses to be as efficient as the original ones, just as in the affine case. By policy, we mean choosing the index transformation itself. We discuss how the analysis can accommodate non-affine index expressions.

Applying a Selected Transformation

First, let us consider the mechanism aspect. Whether array index expressions are affine or not, the transformed access generally involves no more indexing overhead than the original one because the address of an array element is an affine function of the indices in both cases. In the affine case, we have explained how the transformed index vector y, like the original x, can be expressed directly as an affine function of the iteration vector i. This hinges on the associativity of matrix multiplication (see (2.3) on page 18 in Section 2.2). In the non-affine case, a similar argument does not hold. Thus, it might appear that extra computation would be needed to compute y from x when the access is performed at run time.

In fact, no extra computation is required because y need not be computed explicitly. The address of an array element is an affine function of the array indices — original or transformed. Traditionally, the scalar offset of an element `X[x1,x2,...]` in a row-major array X is

$$s = \sum_{j=1}^{m} \left(\prod_{k=j+1}^{m} (u_k - l_k + 1) \right) (x_j - l_j) \tag{3.10}$$

where l_k, u_k, and x_k are respectively the lower bound, upper bound, and array index in the k-th dimension of an m-dimensional array. In other words, the offset s is an affine function of the array index vector x:

$$s = vx + v_0 \tag{3.11}$$

where v is a constant vector and v_0 a constant scalar offset, both dependent only on the array bounds. As we shall see in Chapter 4, after array restructuring, the offset of an element within the restructured array, denoted s', remains an affine function of the transformed index vector y and therefore also of the original index vector x:

$$s' = v'y + v_0' = (v'T)x + v_0' \tag{3.12}$$

Without array restructuring, when performing an access, we may have to compute the array indices in x and then the location s according to (3.11). With array restructuring, we still compute x as before, and then s' according to (3.12). The constants involved in these two cases may be different, but the form of the code should be the same and thus equally efficient because the same compiler optimization techniques should apply in both cases, provided that these techniques do not depend on the particular values of constants. For example, induction variable elimination would still apply unless a particular implementation relies on the increment having specific values, rather than merely the knowledge that the increment is loop-invariant.

Selecting the Index Transformation Matrix

Now we turn to the policy aspect of handling non-affine index expressions: choosing the index transformation itself. We extract as much access pattern information as we can from the index expressions, though perhaps not as much as in the affine case. Any limitation in the analysis, however, does not make the transformed program incorrect; at worst, we mistakenly select a transformation that does not improve performance as we hope. In the following discussion, we first review what information is needed to compute an index transformation and then discuss how far we can obtain that information from non-affine accesses.

To choose an index transformation, we need to know how array indices change as various loop variables are incremented. As discussed before, in the affine case, these changes can be determined precisely: they come from the columns of the access matrices. In the non-affine case, we have less accurate but often still useful information.

Let us look at several examples that illustrate possible techniques before we discuss our approach in general terms. These examples are shown in the upper half of Figure 3.11, in which `f(.)` and so on are non-affine expressions about which we know little except that their values are completely determined by their arguments. For the first access `X[i2,f(i1)]`, the non-affine `f(i1)` prevents us from knowing which column is

```
FOR i1 = ...
  FOR i2 = ...
    X[i2,f(i1)]
```

```
FOR i1 = ...
  FOR i2 = ...
    X[f(i1)+i2,g(i1)+i2]
```

```
FOR i1 = ...
  FOR i2 = ...
    X[f(i1)+h(i2),g(i1)+h(i2)]
```

$$x = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} i_1 \\ f(i_1) \\ i_2 \end{bmatrix} \qquad x = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} i_1 \\ f(i_1) \\ g(i_1) \\ i_2 \end{bmatrix} \qquad x = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} i_1 \\ f(i_1) \\ g(i_1) \\ i_2 \\ h(i_2) \end{bmatrix}$$

Figure 3.11: Examples of Non-Affine Array Accesses

accessed in each outer loop iteration. We do know, however, that every execution of the inner loop accesses a column. Therefore, we should transpose the array regardless of `f(i1)`. For the second, less obvious access `X[f(i1)+i2,g(i1)+i2]`, we know that the inner loop accesses elements on the same diagonal consecutively, although we cannot determine which diagonal it is for any given outer loop iteration. Nevertheless, this partial information is enough for us to choose a transformation, say $\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$, that places elements on such a diagonal consecutively in memory. Consider the third access: `X[f(i1)+h(i2),g(i1)+h(i2)]`. In this case, the inner loop also accesses elements along a diagonal because given the value of `i1`, varying `i2` always changes both array indices by the same amount. We do not know which diagonal it is. Nor can we tell how the elements in that diagonal are accessed: they could be accessed in some irregular order. Despite this lack of information, it still seems beneficial to store elements in the same diagonal consecutively. Even if the inner loop may not go through them consecutively, at least the memory accesses are likely to stay within a smaller region than if the array is stored in the original, row-major order. The performance benefits are uncertain, however.

Generally, our analysis for non-affine accesses identifies non-affine sub-expressions in index expressions and writes each index expression as an affine function of these non-affine sub-expressions and the loop variables, as we have done in the examples above.

Moreover, we recognize non-affine sub-expressions common to different array dimensions, such as the `h(i2)` in `X[f(i1)+h(i2),g(i1)+h(i2)]`. Thus, the three previous examples may be represented by "augmented" access matrices and iteration vectors as shown in the lower half of Figure 3.11. Each column in an augmented access matrix corresponds to a loop variable or one of the non-affine sub-expressions identified, which are themselves functions of loop variables. Although we cannot determine exactly how each non-affine sub-expression varies with the loop variables, we do know which loop variables it depends on. Therefore, when we order columns from individual access matrices to form an aggregate access matrix, we treat a column for a non-affine sub-expression as if it were for the deepest loop whose variable the sub-expression depends on. For instance, the column $\begin{bmatrix}1\\1\end{bmatrix}$ in the third example is deemed to correspond to the inner loop variable `i2`. With this rule, we form the aggregate access matrix as before, ordering columns by the depths of their corresponding loops and with tie-breaking rules aimed at preserving the original array layout as much as possible. The index transformation matrix is then chosen just as discussed earlier.

How do we identify these non-affine sub-expressions? Roughly speaking, we examine the expression trees for array indices from the bottom up. If a node is an affine function of loop variables and already identified non-affine sub-expressions, it is annotated as "affine" and with the coefficients of the affine function. Specifically, a node that adds or subtracts two affine nodes or multiplies a loop-invariant constant to an affine node falls into this category. The parent's coefficients are computed from the children's. In other cases, the node is annotated as non-affine, and we record the set of loop variables it depends on, which is the union of the sets for its children. This procedure is similar to value numbering [Alpern, Wegman, and Zadeck 1988; Bacon, Graham, and Sharp 1994; Reif and Lewis 1986].

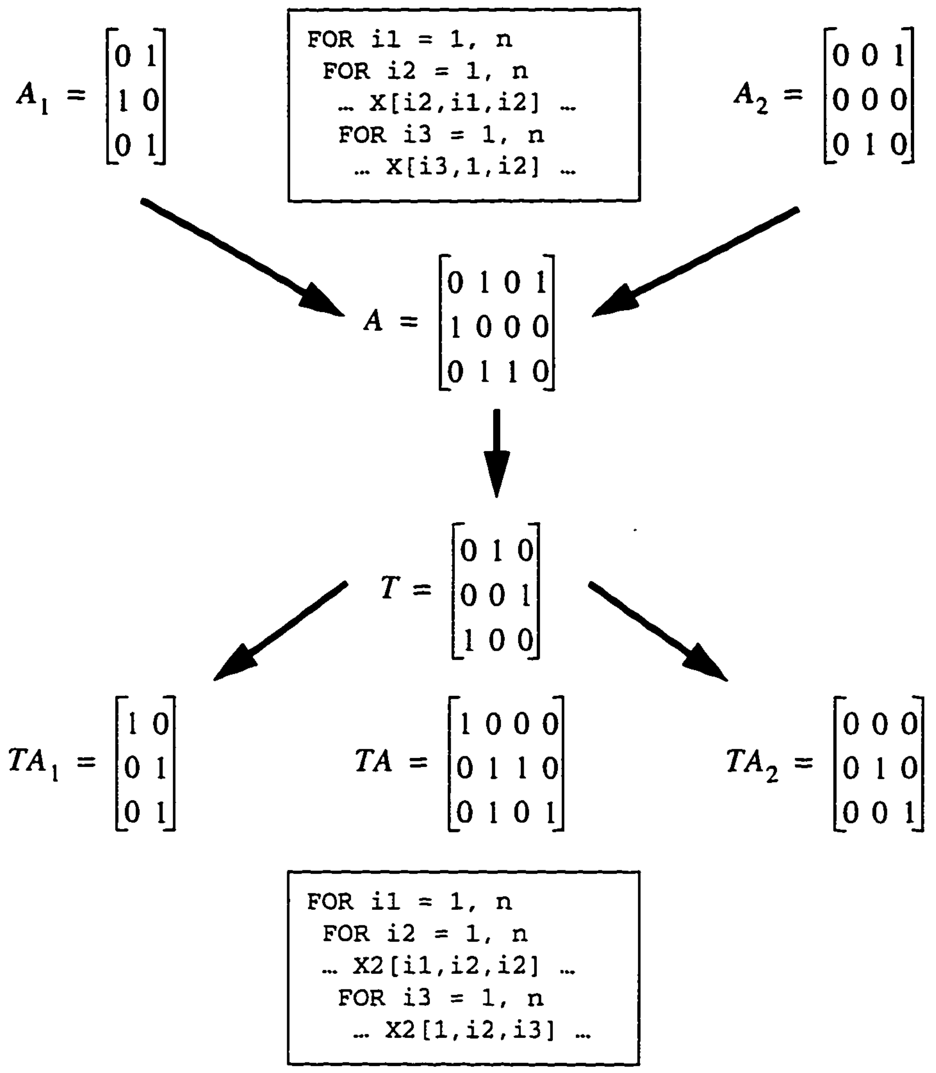
Summary

In summary, non-affine array accesses by no means hamper the application of a selected index transformation. The transformed array accesses do not involve any more indexing overhead than the original ones because the address computation and index transformation, both represented by affine functions, can be combined. Non-affine accesses do, however, complicate the analysis for choosing the transformation. We try to express non-affine index expressions as affine functions of non-affine sub-expressions to determine, as far as we can, how array indices change when loop variables are incremented and thus choose a suitable index transformation.

3.3.3 Imperfect Loop Nests

Imperfect nesting of loops often complicates, if not hampers, loop restructuring. For a perfect loop nest, one can think of many loop transformations (e.g., interchange [Allen and Kennedy 1984], reversal [Wedel 1975], and skewing [Wolfe 1989b]) simply as the reordering of indivisible iterations. They can be expressed formally as transformations of an iteration space whose points represent iterations of the innermost loop. For an imperfect loop nest, however, iterations of non-innermost loops also have to be represented in the same framework. Work on this continues [Kelly and Pugh 1995; Pugh 1991]. Alternatively, the compiler may split an imperfect nested loop into equivalent, perfect loop nests with loop distribution, transform the latter individually, and finally recombine them with loop fusion if appropriate [Kennedy and McKinley 1994].

On the other hand, array restructuring can easily handle imperfect loop nests. Since the loop structure is unchanged, as before we only need to compute the changes in array indices as various loop variables are incremented. These changes can be obtained from columns of the access matrix as before. The access matrix for an access outside the innermost loop may have fewer columns than one within, but all access matrices have the same number of rows, which is the number of dimensions for the array in question. Therefore, access matrix columns may be extracted and ordered according to loop levels to form an aggregate access matrix just as discussed before. This is illustrated by the example in Figure 3.12. Despite the different dimensions of access matrices A_1 and A_2, this example is handled in much the same way as the perfect loop nest in Figure 3.10 on page 53.



```
FOR i1 = 1, n
 FOR i2 = 1, n
  ... X[i2,i1,i2] ...
  FOR i3 = 1, n
   ... X[i3,1,i2] ...
```

$$A_1 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad A_2 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

$$A = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

$$T = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

$$TA_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \qquad TA = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \qquad TA_2 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

```
FOR i1 = 1, n
 FOR i2 = 1, n
 ... X2[i1,i2,i2] ...
  FOR i3 = 1, n
   ... X2[1,i2,i3] ...
```

Figure 3.12: Handling an Imperfect Loop Nest

3.3.4 Multiple Loop Nests

We have so far focused on optimizing for a single loop nest. Before loop execution, elements of the original array are copied to the restructured array; during loop execution, the restructured array is accessed instead of the original; after loop execution, elements are copied back to the original array if the array is written. The whole process is then repeated for the next loop nest. This strategy is simple but potentially inefficient: the copying overhead may outweigh the performance gain in loop execution.

This copying overhead can be amortized if a single array transformation is chosen for a larger segment of the program that contains more than one loop nests. The array needs to be restructured only before and, if written, after the entire set of loop nests; the intermediate copying is eliminated. We can trivially extend the analysis techniques discussed so far for this purpose: simply enclose the program segment in question with an imaginary loop that has only one iteration and then analyze the resulting imperfect loop nest as before.

The nontrivial issue that warrants more discussion is which loop nests should be grouped together. When making this decision, we have to consider the tradeoff between copying overhead and potential improvement in loop execution performance. At one end of the spectrum, choosing an array transformation best for each loop nest would maximize the performance gain but at a substantial copying cost; at the other, grouping as many loop nests together as possible would amortize the overhead but may cause some loop nests to be executed with an array layout unsuitable for the access pattern. In the following, we first look at several options, going from fine- to more coarse-grained grouping, and then discuss the approach we have taken.

First, we can consider each loop nest individually. Arrays are restructured before and, if written, after each loop execution. We have already discussed this option.

Second, we may consider a "basic block" of loop nests at a time. By a "basic block," we mean a sequence of loop nests (perhaps with intervening non-loop code) that are always executed together, analogous to a basic block of instructions. As discussed above, we can handle this simply by enclosing the loop nests inside an imaginary loop with one iteration.

We need not always group the longest straight-line sequence of loop nests together. Instead, we could break such a sequence into subsequences of loop nests such that each subsequence has its own array transformation and the array is dynamically restructured between subsequences. The sequence can be broken up in different ways for different arrays; there is no reason why all arrays must be restructured at the same points in program execution. For each array, the access patterns of loop nests in a subsequence should be similar enough that there is some array transformation appropriate (though not necessarily ideal) for all of them. While assigning each loop nest into a separate subsequence no doubt achieves this, on the other hand we want to minimize the number of subsequences or, more precisely, the copying overhead between subsequences. This problem can probably be solved with some form of dynamic programming. We would need cost models to estimate the copying overhead and the performance benefit of a given array transformation to a given series of loop nests, together with an algorithm that chooses array transformations to maximize such benefits. Much remains to be done, but we have not explored this direction further.

Third, we may consider an entire procedure. Arrays are restructured dynamically only at procedure calls and returns. Interprocedural analysis is not required. This is especially appropriate for procedures that are intended for a library and therefore may be called by code that has been compiled separately without any notion of array restructuring. Arrays are stored in a canonical form whenever control is being transferred between procedures. Alternatively, if the calling convention includes the layout information of arrays passed between caller and callee, arrays can be restructured only as required even across procedure calls.

Finally, we can consider the entire program and choose a single transformation for each array. Although this approach requires sophisticated interprocedural analysis to locate all accesses, including aliased accesses, to each array, it has the major advantage of eliminating not only all runtime copying overhead but also the need for extra memory to store the restructured arrays in addition to the originals. However, there may not be a single array transformation appropriate throughout the entire program. If there is an obvious choice, the programmer may have written the program based on that to begin with.

Our strategy for multiple loop nests combines compiler analysis on each loop nest individually with "lazy restructuring" at run time. The compiler chooses a transformation for each array and each loop nest, and transforms the accesses accordingly. However, we do not obliviously copy from and to an original array laid out in some fixed canonical order. Instead, our runtime system caches valid copies of an array, each corresponding to a different transformation. Before a loop nest is executed, it looks for a valid copy matching the desired transformation for this loop nest, as determined by the compiler. If such a copy is found, it is used; otherwise, the runtime system creates one from a valid copy. After loop execution, the restructured array is preserved, and other copies are invalidated if the array has been written. This is analogous to the maintenance of cache coherence in shared-memory multiprocessors with an invalidation-based protocol. For compatibility with separately-compiled procedures, arrays are assumed to be in row-major order on procedure entry and restored to row-major order on exit. In other words, array restructuring occurs only within a procedure.

The major limitation of this strategy is the extra storage required to keep multiple copies of the same array. Keeping more than two copies does not imply proportionally increased pressure for cache space because only some of them are accessed simultaneously — two during copying, one in loop execution. However, for some fixed capacity of storage (be it memory or disk), keeping extra copies does limit the size of the array (and hence the problem) that can be handled. Therefore, for large problems whose bottleneck is memory or disk capacity, we should discard some (perhaps all but one) of the copies.

When an array should be restructured is as important and difficult a question as how. The answer depends on a tradeoff between allowing each loop nest to run with the best array layouts and the runtime copying overhead to create those layouts. Another concern is the extra storage needed for multiple copies of an array, which may limit problem sizes for a fixed amount of available storage. In this section, we have explored only some of the possible options. In fact, we have not even touched on the possibility of restructuring an array within a loop nest, rather than always before and after it. Much more work is needed to fully understand the issue of when to restructure an array.

3.4 Summary

To improve the spatial locality of a single array access by array restructuring, we find a unimodular index transformation matrix with a, roughly speaking, "lower-triangular" non-zero structure. Our procedure for this consists of two steps. First, a Gaussian elimination-like algorithm transforms the access matrix to the right form with elementary row operations, accumulating the effects of those operations in a nonsingular, but not necessarily unimodular, matrix. The algorithm always produces a transformed access matrix with identical rank and height profiles; we have shown that it is impossible to flatten the access matrix columns any further. In the second step, a unimodular matrix is computed from the nonsingular matrix just obtained by means of the latter's Hermite normal form.

We have also discussed handling general access patterns. If there are multiple accesses to one array, the same algorithm for computing the transformation matrix can be applied to an aggregate access matrix constructed from the columns of the individual ones. Non-affine index expressions do not hinder the application of an already selected index transformation. They do make choosing that transformation harder by making the analysis less precise than in the affine case. We have nonetheless developed techniques to gather some useful access pattern information that guides the choice. Imperfect loop nests pose no special problem. Finally, using one array transformation for multiple loop nests can amortize the runtime copying overhead but involves complex tradeoffs that require further study.

Chapter 4

Linearizing Restructured Arrays

What does it mean to store elements of an array in row-major order? In Chapter 2, we assume that the restructured array is row-major. Accessing an element involves, conceptually, computing the original index vector, applying a linear transformation, and finally indexing the restructured array under this assumption. Based on the same assumption, we have discussed in Chapter 3 how we can relate locality to the access matrix and thus choose a suitable index transformation to enhance locality. To see what this key assumption means to the restructured array, we first look at what it means traditionally.

Traditionally, an array declaration specifies lower and upper bounds on each array index. The bounds are constant: they do not depend on the values of other indices. Thus, for a row-major, m-dimensional array, finding an element with indices $x_1, \ldots, x_m$ requires computing a scalar offset s into the array as

$$s = \sum_{j=1}^{m} \left(\prod_{k=j+1}^{m} (u_k - l_k + 1) \right) (x_j - l_j) \tag{4.1}$$

where l_k and u_k are respectively the lower and upper bounds of the k-th array index. This can also be expressed in vector form:

$$s = vx - vl \tag{4.2}$$

where

$$v_j = \prod_{k=j+1}^{m} (u_k - l_k + 1) \qquad 1 \le j \le m \tag{4.3}$$

We call the row vector v the *linearization vector.* The address of the element can be further computed as an affine function of the offset s, using the size of an element and the starting address of the array. Since only affine functions are used, a compiler can optimize the address calculation by strength reduction: replace the brute force evaluation of affine functions, which involves multiple integer multiplications and additions per access, with incremental evaluation, which only involves incrementing an address pointer by a precomputed amount in each iteration.

Thus, to lay out elements of the restructured array in row-major order, we need to compute the same two pieces of information: the bounds of the restructured array and, from those bounds, the linearization vector. This is simple if our index transformation just permutes array indices (as in Section 3.2.1). We can obtain the transformed array bounds by permuting the original array bounds accordingly. The linearization vector is then calculated in the traditional way from the transformed (i.e., permuted) array bounds.

The rest of this chapter considers the general case where the index transformation matrix is an arbitrary unimodular matrix. Computing the bounds of the restructured array is relatively simple. It is discussed in Section 4.1. Computing the linearization vector from those bounds, however, is much harder than in the traditional case because those bounds may not be constant. In other words, the lower and upper bounds for one array index may depend on the values of other array indices. Such non-constant array bounds render the traditional solution described above inapplicable. Section 4.2 explains this problem more clearly. Section 4.3 introduces our solution with an example. Section 4.4 and Section 4.5 discuss the general algorithm in detail.

4.1 Computing the Bounds of the Restructured Array

To compute the bounds of the restructured array, we express those of the original array algebraically, as well as geometrically, and transform these representations with simple linear algebra.

We can represent the bounds of the original array algebraically as a set of inequalities involving the array indices, or geometrically as a convex polyhedron in the array index space. Both representations are illustrated on the left of Figure 4.1 for the two-dimensional array X shown at the top of the diagram.

- In the algebraic representation, each inequality simply states that some array index must be at least its lower bound, or at most its upper bound. To facilitate mathematical manipulation, we write the inequalities in vector form as

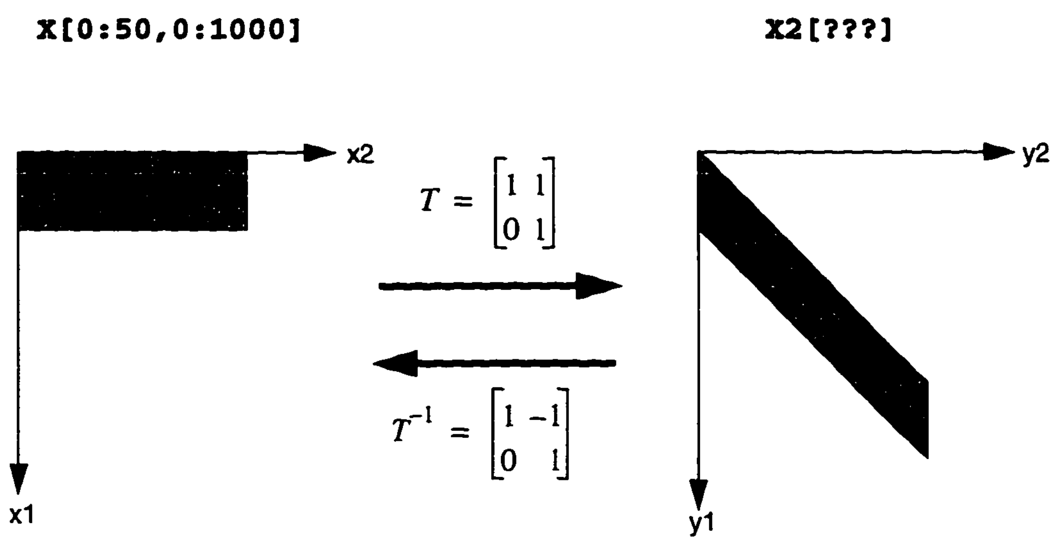
  $$Bx \geq b \tag{4.4}$$

  in which the comparison is interpreted row by row, with each row of the matrix B and vector b corresponding to an inequality in standard form.

- In the geometrical representation, valid array index vectors correspond to integral points (i.e., points whose coordinates are integers) within a convex polyhedron in the index space. Each face of the polyhedron is a hyperplane defined by one of the inequalities, and points in the polyhedron have coordinates that satisfy all the inequalities. For the original array, the polyhedron is simply a rectilinear region (a rectangle in the two-dimensional case) because all the inequalities involve only one array index.

Given the index transformation matrix T, we can compute the bounds of the restructured array by simple substitution. Since T is unimodular, it is nonsingular. Thus,

$$x = T^{-1}y \tag{4.5}$$


X[0:50,0:1000]
X2[???]
x2
x1
$T = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$
$T^{-1} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}$
y2
y1

$x_1 \geq 0 \qquad x_1 \leq 50$

$x_2 \geq 0 \qquad x_2 \leq 1000$

$y_1 \geq y_2 \qquad y_1 \leq y_2 + 50$

$y_2 \geq 0 \qquad y_2 \leq 1000$

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \geq \begin{bmatrix} 0 \\ -50 \\ 0 \\ -1000 \end{bmatrix} \qquad \begin{bmatrix} 1 & -1 \\ -1 & 1 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \geq \begin{bmatrix} 0 \\ -50 \\ 0 \\ -1000 \end{bmatrix}$$

$$Bx \geq b \qquad y = Tx \qquad \left(BT^{-1}\right)y \geq b$$

Figure 4.1: Transforming Array Bounds

where x and y are index vectors for the original and restructured arrays respectively. Substituting this back into (4.4), we get

$$\left(BT^{-1}\right)y \geq b \tag{4.6}$$

This gives us the linear inequalities that the transformed array indices must satisfy if the original ones satisfy (4.4). The right half of Figure 4.1 shows the transformed inequalities for the example. Notice that the transformed array bounds cannot be expressed as a conventional array declaration because the bounds on one array index may depend on the values of the others. In geometrical terms, each face of the polyhedron in the original index space is mapped through the index transformation to a hyperplane in the transformed index space. These image hyperplanes together delineate the image polyhedron representing the bounds of the restructured array.

All integral vectors within the transformed array bounds (4.6) correspond to original index vectors within the original array bounds (4.4), and therefore identify elements that the restructured array must contain. This is because any integral vector y satisfying (4.6) is the image of $x = T^{-1}y$, which is integral (since both T and T^{-1} are unimodular) and satisfies (4.4). Similarly, the converse is also true.

4.2 Problem of Non-Constant Array Bounds

Having computed the bounds of the restructured array according to (4.6), we lay out the elements (in row-major order) in such a way that the location of an element can be efficiently computed from the array indices. Specifically, we find an affine function to express the location in terms of the array indices. This affine function is represented by the linearization vector. In this section, we explain why computing this vector is nontrivial in our case, highlighting the difficulty with a simple but unsatisfactory solution.

Computing the linearization vector is complicated by array restructuring. When array bounds do not depend on array indices, as in the traditional case, (4.1) gives a well-known, closed-form answer. However, in general our array restructuring technique may lead to array bounds that vary with other array indices, as Figure 4.1 has shown. In geometrical terms, the bounds of the restructured array are delineated by a parallelepiped[1], rather than

1. A parallelepiped is the analogue of a parallelogram in three or more dimensions.

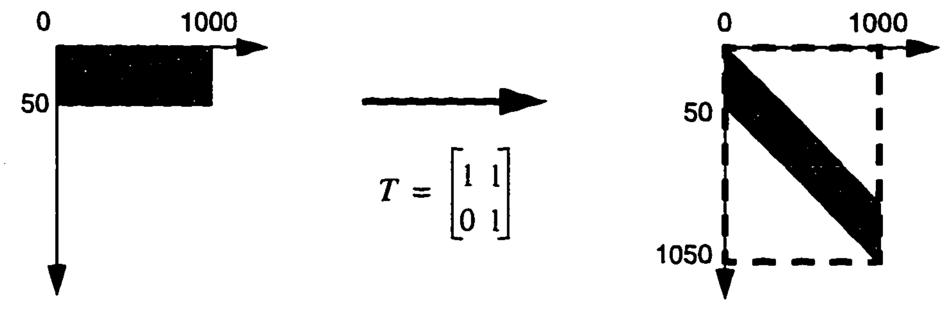


Figure 4.2: Laying Out Elements of Restructured Array — Tentative Solution

a rectilinear region, in the array index space. The traditional method therefore does not apply.

A simple but unsatisfactory solution is illustrated by Figure 4.2, showing the same example as Figure 4.1. As in Figure 4.1, the rectangle on the left and the parallelogram on the right represent the bounds of the original and restructured arrays respectively. The tentative solution uses the dashed box on the right as the bounds of an "expanded" array. In other words, we compute the two extremities of the transformed array bounds (the shaded parallelogram) in each dimension. Then, we conventionally declare an array with the extremities as lower and upper bounds. Since this expanded array contains all the elements necessary for storing the original, it can be used in place of the restructured array.

However, this naive solution could waste a large, potentially unlimited, amount of memory just to make it efficient to find an element given the array indices. In Figure 4.1, this wasted memory corresponds to the two empty triangles in the dashed box. In this example, the original array has 51,051 elements (51 rows by 1001 columns), but the naively expanded array has 1,052,051 elements (1051 rows by 1001 columns). Only one-twentieth of the expanded array is used! This utilization rate, already poor, can be made arbitrarily small by making the original array, and hence also the restructured array, "skinnier." Moreover, the unused memory locations are interspersed with the utilized ones at a moderately fine granularity: each contiguous range of unused memory spans roughly one

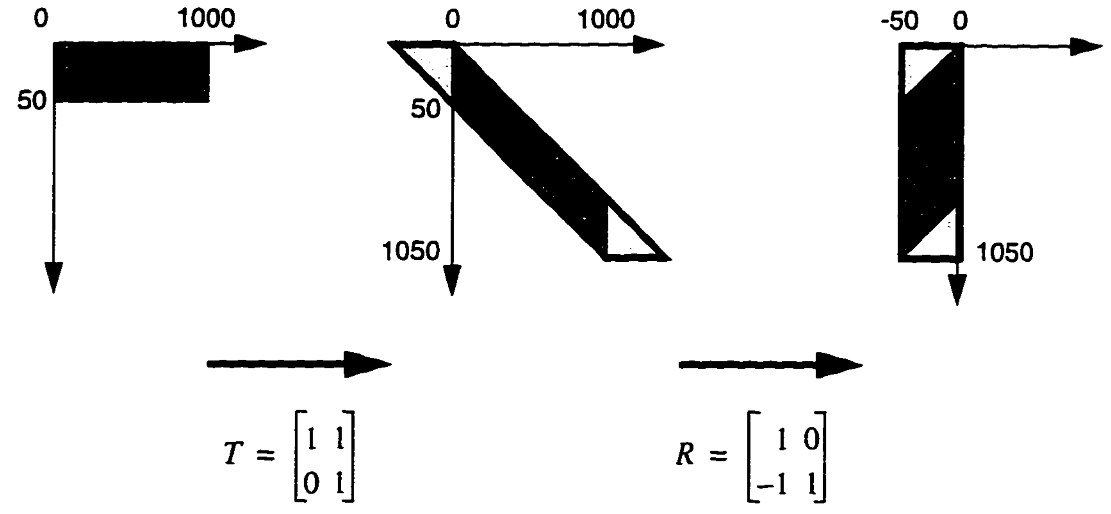

0
1000
50
0
1000
50
1050
-50
0
1050
$T = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$
$R = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}$

Figure 4.3: Laying Out Elements of Restructured Array — Our Solution

row of elements. Therefore, we cannot reduce the waste by not allocating physical pages for the unused portions. The same problem, of course, occurs also in higher-dimensional cases.

4.3 Introducing Our Solution

We address this problem with a two-step algorithm that, roughly speaking, finds a transformation to turn the array bounds back into constants without jeopardizing prior transformations for better locality. Once we have constant array bounds, the traditional method can be used to compute the linearization vector. Figure 4.2 shows how the algorithm applies to the example. Again, the rectangle on the left and the deeply shaded parallelogram in the middle represent the bounds of the original and restructured arrays respectively.

1. The first step relates to the middle part of the figure. It computes a parallelogram (the shaded, outer parallelogram in the middle) with a pair of horizontal sides (the

top and bottom sides) to enclose the parallelogram representing the bounds of the restructured array (the deeply shaded, inner parallelogram). Why horizontal sides are significant will become clear below. The outer parallelogram defines the bounds of an expanded array that contains all the necessary elements (corresponding to integral points in the inner parallelogram) and perhaps some unnecessary ones. Naturally we want to minimize the number of the latter. Hence, the outer parallelogram should enclose the inner one as tightly as possible.

2. The second step takes the middle part of the figure to the right part. It computes a linear transformation, represented by matrix *R*, to transform the outer parallelogram to a rectangle. This transformation in effect shifts each row leftward by the row index (i.e., row 0 at the top is not moved, row 1 is shifted left one column, and so on). With this transformation, the horizontal sides remain horizontal, whereas the slanted sides are made vertical by the horizontal movement. The resulting expanded array has constant array bounds. Thus, it can be handled traditionally.

The properties of the additional transformation *R* have important implications. First, because it is linear, it can be combined with the affine functions that represent other intermediate steps of computing the location of the accessed element from the loop variables. No extra indexing overhead is required.

Second, by only shifting rows horizontally, *R* does not jeopardize the previous transformation *T* chosen to improve spatial locality. Earlier on, we have described how to choose *T* so that the innermost loop goes through the array index space horizontally, rather than diagonally, as the arrows in the figure show. Since *R* shifts rows horizontally, it keeps a horizontal direction horizontal and thus preserves the effect of *T*.

Finally, it should be noted that we have not eliminated unused memory. We still allocate but do not use the memory corresponding to the two lightly shaded triangles in the right part of Figure 4.2. In this example, the expanded array has 53,601 elements (1051 rows by 51 columns), of which about 5 percent are not used. In general, at most half the allocated memory is unused in the two-dimensional case. A bound for any number of dimensions is proved in Appendix A.

4.4 Computing the Linearization Vector

Having seen how a specific example is handled, we can now more easily discuss the full details of the general algorithm. We have to find a linearization vector v, for the restructured array, that meets the following requirements.

- v is integral. This facilitates address computation and the associated compiler optimizations.
- v represents a row-major storage order — the storage order of the restructured array assumed throughout the earlier analysis. In other words, if transformed index vectors y and y' are both within the transformed array bounds and y is lexicographically less than[2] y', then $vy < vy'$. This subsumes the obvious requirement that two different vectors within the bounds must lead to different offsets, because one of the these two vectors must be lexicographically less than the other.

Among vectors meeting these requirements, we naturally want to find one that minimizes the amount of allocated memory. The linearization vector given by (4.3) satisfies both requirements and uses the minimum amount of memory. However, as noted earlier, this formula applies only to constant array bounds, while our transformed bounds may not be constant. Therefore, we need a more general way of computing the linearization vector.

Our overall approach is to relax the *transformed bounds* so that the resulting bounds, which we call the *relaxed bounds*, are in a special form for which we know how to com-

2. A vector is lexicographically positive (negative) if its leading nonzero is positive (negative). A vector y is lexicographically less than, or before, a vector y' if $y - y'$ is lexicographically negative. This means that in the first dimension where the two vectors differ, the component of y is less than that of y'. "Lexicographically greater than" or "lexicographically after" are defined similarly.

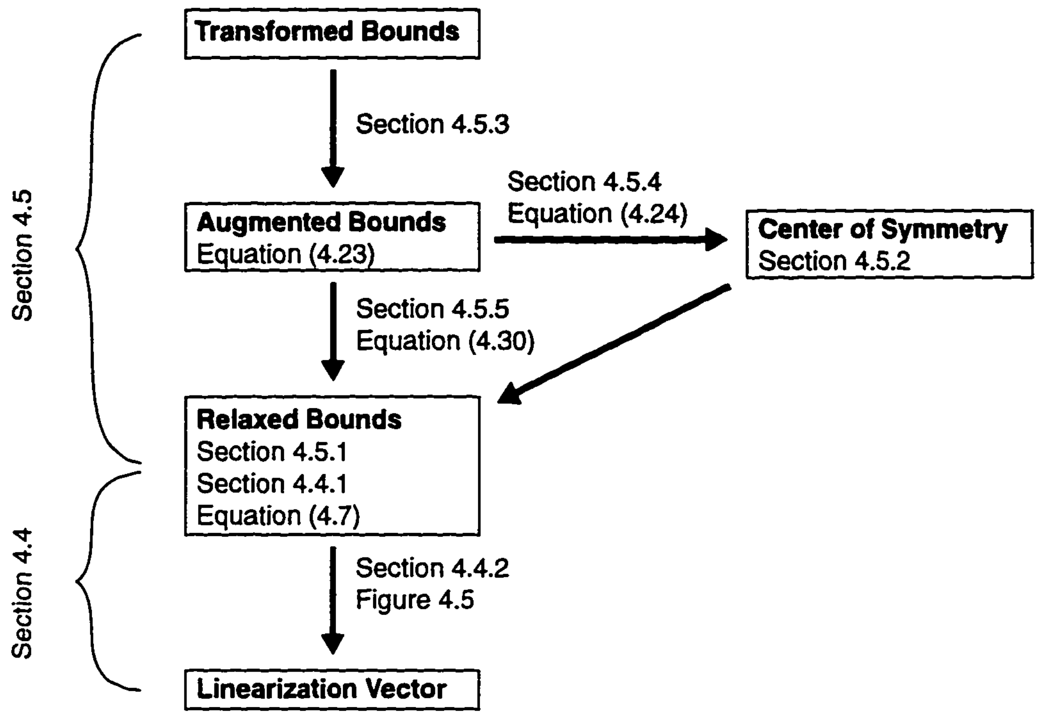


Figure 4.4: Computing the Linearization Vector: A Road Map

pute the linearization vector. In geometrical terms, we look for a special form of polyhedron to enclose the polyhedron representing the transformed bounds. An unsatisfactory application of this approach is the "dashed box" solution attempted in Section 4.2: we relaxed the transformed bounds directly to a set of constant bounds, but found it too wasteful in the use of memory.

This section and the next present a much better solution. Figure 4.4 outlines the whole process and indicates where related discussion is. In this section, we first specify the form of the relaxed bounds and explain why it makes computing the linearization vector easier. Then, we describe the algorithm to calculate the linearization vector and argue that it produces a vector meeting our requirements. This roughly corresponds to the second step (i.e., transforming the outer parallelogram and then linearizing the resulting rectangle in Figure 4.2 on page 70) in the previous section's example. The issue of computing the relaxed bounds themselves from the transformed bounds is left to the next section. It roughly corresponds to the first step (i.e., finding an enclosing parallelogram with a pair of horizontal sides) of the example. The discussion is organized in this way because knowing how the relaxed bounds are used is crucial to understanding why they are computed in the way to be described later.

4.4.1 Relaxed Bounds

We have defined the relaxed bounds as the result of relaxing the transformed bounds to a special form that facilitates computing the linearization vector. We now specify exactly what that form is and explain how it helps the calculation of the linearization vector.

Intuitively, we need a special form such that the relaxed bounds in the transformed index space, which are not constant but in this form, can be further transformed through another linear transformation to bounds that are constant. (As Section 4.3 explains, in two dimensions, geometrically this special form is "a parallelogram with a pair of horizontal sides.") With the resulting bounds, we can compute a linearization vector in the traditional way (since those bounds are constant) and combine this vector with all prior transformations by composing the corresponding affine functions. Furthermore, we require that the extra transformation here preserve the locality improvement effected by the previous transformation.

Formally, we require that the relaxed bounds — a set of inequalities in the transformed array indices — can be expressed in vector form as

$$r_l \le Ry \le r_u \qquad (4.7)$$

where R is an $m \times m$ matrix required to be lower-triangular with a unit diagonal (i.e., all diagonal elements must be 1), r_l and r_u are m-dimensional vectors, and y denotes the transformed index vector as before. For the example in Figure 4.2, the relaxed bounds are represented by the shaded, outer parallelogram in the middle, and the transformed bounds by the deeply shaded, inner parallelogram. The special form stipulated here translates into the requirement for "a pair of horizontal sides" in that example. See Section 4.5.1 for the geometrical implications in higher-dimensional spaces.

If the relaxed bounds are in this form, we can transform the transformed index vector y again, with the matrix R, to another vector space so that the resulting bounds in this space are constant, allowing them to be handled traditionally. Formally, we can define

$$z = Ry \tag{4.8}$$

In Figure 4.2, this transformation corresponds to further transforming the outer parallelogram to a rectangle. Substituting (4.8) into (4.7) yields the bounds on z, which are

$$r_l \le z \le r_u \tag{4.9}$$

In Figure 4.2, r_l and r_u represent the four sides of the rectangle on the right. We might then compute a linearization vector, denoted w, for these constants bounds in the traditional manner. The linearization vector v, which takes a transformed index vector y to an element's offset into the restructured array, is therefore equal to wR.

Because of the form R is required to have, the extra transformation preserves the effect of the transformation applied earlier for better locality: since R is lower-triangular with a unit diagonal, left-multiplying it to an access matrix changes neither the height nor the value of the top nonzero of any access matrix column. In this sense, prior locality optimizations are preserved. In particular, an access matrix column representing a unit stride through memory consists of leading zeros and a single 1 or -1 in the last dimension, and this is not affected by the application of R.

4.4.2 Computing Linearization Vector from Relaxed Bounds

Computing the linearization vector v from the relaxed bounds in (4.7) is in fact more complicated than just described because we cannot assume that R, r_l, and r_u are always integral. If R is not integral, the vector $z = Ry$ may be non-integral for an integral transformed index vector y. Thus, restructured array elements may be identified by *non-integral* vectors z within the bounds $r_l \le z \le r_u$. Because of this and possibly non-integral r_l and r_u, strictly speaking the linearization vector w cannot be computed traditionally after all.

For this reason, instead of first transforming the relaxed bounds to constant bounds and then finding w with the formula (4.1), we integrate the two steps in one algorithm. Whether R, r_l, and r_u are integral or not, we compute a vector w such that the product $v = wR$ is integral, even though w itself may or may not be integral. The final result is v — the linearization vector taking transformed index vectors y to offsets in the restructured array. Since all the intermediate steps of addressing an element given the original array indices are eventually merged into a single affine function anyway, all that matters is that $vT = wRT$ is integral; non-integral w or R is not a problem.

The algorithm is shown in Figure 4.5. We discuss this algorithm in the remainder of this subsection. We show that the linearization vector v it computes does meet the requirements stipulated at the beginning of this section. Furthermore, the computed v, to a good approximation, requires the least amount of memory among all vectors that also satisfy those requirements. Finally, we show that the algorithm reduces to the traditional method in the simple case of constant bounds.

First, the algorithm guarantees that v is integral. To see this, consider each component of v in turn. From $v = wR$, we have

$$v_j = \sum_{k=1}^{m} w_k R_{kj} = w_j + \sum_{k=j+1}^{m} w_k R_{kj} \qquad (4.10)$$

because R is lower-triangular with a unit diagonal. For $j = m$, we know that v_m is equal to w_m, which is set to 1 in the algorithm. Thus, v_m is an integer. As for $1 \le j < m$, when the algorithm chooses w_j, sum is precisely the rightmost summation in the above equa-

```
R is an $m \times m$ lower-triangular matrix
$r_l$ and $r_u$ are m-dimensional column vectors
v and w are m-dimensional row vectors

$w_m = 1$

FOR $j = m-1$ DOWNTO 1 DO

  at_least = $(r_{u,j+1} - r_{l,j+1} + 1)\, w_{j+1}$

  sum = $\sum_{k=j+1}^{m} w_k R_{kj}$
  $w_j$ = smallest number >= at_least and with
        fractional part = ceiling(sum) - sum

ENDFOR

$v = wR$
```

Figure 4.5: Computing Linearization Vector from Relaxed Bounds

tion. Choosing w_j, which can be shown to be positive by induction on j, such that its fractional part is `ceiling(sum) - sum` ensures that

$$w_j - \lfloor w_j \rfloor = \left\lceil \sum_{k=j+1}^{m} R_{kj} w_k \right\rceil - \sum_{k=j+1}^{m} R_{kj} w_k \qquad (4.11)$$

and hence

$$v_j = w_j + \sum_{k=j+1}^{m} w_k R_{kj} = \lfloor w_j \rfloor + \left\lceil \sum_{k=j+1}^{m} R_{kj} w_k \right\rceil \qquad (4.12)$$

which is an integer because both terms in the rightmost expression are results of floor and ceiling functions. Hence, the linearization vector v is integral.

Second, if y and y' are both within the relaxed bounds and y is lexicographically less than y', then $vy < vy'$. Since $vy' - vy = wRy' - wRy = w(Ry' - Ry)$, we start justifying this claim by considering the possible ranges for the components of $Ry' - Ry$, which is denoted d from now on.

- The leading nonzero of d is at least 1. Since R is lower-triangular, the leading nonzero of $d = R(y' - y)$ is in the same dimension as that of $y' - y$. In fact, the two leading nonzeros are equal since all diagonal elements of R are 1. As $y' - y$ is lexicographically positive, its leading nonzero is positive. Moreover, as $y' - y$ is integral, this leading nonzero is at least 1.

- d_j is at least $-(r_{uj} - r_{lj})$. Since y and y' are within the relaxed bounds, the j-th components of Ry and Ry' are both between r_{lj} and r_{uj} (see (4.7) on page 75). Their difference, d_j, cannot be less than $-(r_{uj} - r_{lj})$.

Assume that the leading nonzero of d is in the l-th dimension. We would have

$$vy' - vy = wRy' - wRy = wd = w_l d_l + \sum_{k=l+1}^{m} w_k d_k \tag{4.13}$$

Noting the two previous observations on the range for d_l and d_k, we see that

$$\left(w_l d_l + \sum_{k=l+1}^{m} w_k d_k \right) \geq \left(w_l - \sum_{k=l+1}^{m} (r_{uk} - r_{lk}) w_k \right) \tag{4.14}$$

The right-hand side must be at least w_m because the algorithm always chooses w_j to be at least $(r_{u,j+1} - r_{l,j+1} + 1) w_{j+1}$. Specifically, by repeated substitution

$$\begin{aligned}
&w_j \geq (r_{u,j+1} - r_{l,j+1}) w_{j+1} + w_{j+1} \\
&w_j \geq (r_{u,j+1} - r_{l,j+1}) w_{j+1} + w_{j+2}(r_{u,j+2} - r_{l,j+2}) + w_{j+2} \\
&\quad \ldots \\
&w_j \geq w_m + \sum_{k=j+1}^{m} (r_{uk} - r_{lk}) w_k
\end{aligned} \tag{4.15}$$

Combining the equation (4.13) and inequalities (4.14) and (4.15), we thus conclude that $vy' - vy \geq w_m = 1 > 0$.

Thus, we see that the algorithm computes a linearization vector that has the required properties. Among the many vectors having these properties, we want one that minimizes the amount of memory allocated. To this end, the algorithm chooses the smallest allowable value for every w_j. In fact, selecting the smallest possible w_j even has the side-effect of decreasing the value that subsequently chosen w_k $(k = j-1, \ldots, 1)$ is required to exceed (i.e., the value of `at_least`).

Although choosing the smallest possible value for each w_k does not guarantee the minimum amount of allocated memory, it does tend to reduce that amount because the restructured array needs a contiguous memory region for, roughly,

$$w(r_u - r_l) + 1 = \sum_{k=1}^{m} w_k (r_{uk} - r_{lk}) + 1 \tag{4.16}$$

elements. We now justify this approximation. Since the elements are laid out in the lexicographic order of their index vectors (in short, row-major order), we need memory for $vy_{last} - vy_{first} + 1$ elements, where y_{first} and y_{last} are respectively the lexicographically least and greatest vectors within the transformed bounds and thus identify the first and last elements of the restructured array. This size equals $w(Ry_{last} - Ry_{first}) + 1$.

To see that this is approximated by (4.16), we relate the bracketed expression to r_l and r_u. Because R is lower-triangular with a unit diagonal, $R(y' - y)$ is lexicographically positive if $(y' - y)$ is: the leading nonzero of $R(y' - y)$ is in the same dimension as that of $(y' - y)$ and is positive because it equals the latter, which is positive. In other words, if y is lexicographically less than y', the same is true of Ry and Ry'. Hence, Ry_{first} and Ry_{last} are respectively the lexicographically least and greatest in the set of vectors Ry such that y is integral and within the relaxed bounds (4.7). If we do not insist on y being integral, the set of vectors Ry such that y is within the relaxed bounds (in the form

$r_l \le Ry \le r_u$) is simply the set of vectors z such that $r_l \le z \le r_u$. The lexicographically least and most among them are r_l and r_u respectively. Thus, by dropping the integral condition, we informally arrive at the above approximation:

$$w(Ry_{last} - Ry_{first}) + 1 \approx w(r_u - r_l) + 1 \tag{4.17}$$

Finally, note that although our algorithm is more general than the traditional method, it reduces to the latter in the case of constant bounds. For constant bounds, $R = I$ and hence $v = w$. The algorithm always chooses w_m to be 1. As for $1 \le j < m$, notice that sum is always zero because $R_{kj} = 0$ for $k \ne j$. Therefore, the algorithm simply chooses for w_j the smallest integer greater than or equal to at_least. The value of at_least is itself an integer because $r_{l,j+1}$ and $r_{u,j+1}$, being simply the array bounds, are integers. Thus, w_j is simply chosen to be the value of at_least. To sum up, we have

$$\begin{aligned} v_m &= w_m = 1 \\ v_j &= w_j = (r_{u,j+1} - r_{l,j+1} + 1)\, w_{j+1} \qquad 1 \le j < m \end{aligned} \tag{4.18}$$

These recursive equations lead to the closed-form formula in (4.3).

4.4.3 Summary

We have discussed how we compute a linearization vector that satisfies the requirements at the beginning of this section. Essentially, we relax the transformed array bounds so that the resulting bounds have the form specified in (4.7). This allows us to compute a linearization vector that preserves the locality improvement brought about by the earlier index transformation. Figure 4.5 shows our algorithm to do this. The algorithm guarantees that the linearization vector is integral and represents a row-major storage order. It reduces to the traditional method in the simple case of constant array bounds. Moreover, it minimizes the approximate amount of memory required for the restructured array — for the given set of relaxed bounds. In the next section, we discuss how the relaxed bounds themselves are calculated from the transformed bounds.

4.5 Relaxing the Transformed Array Bounds

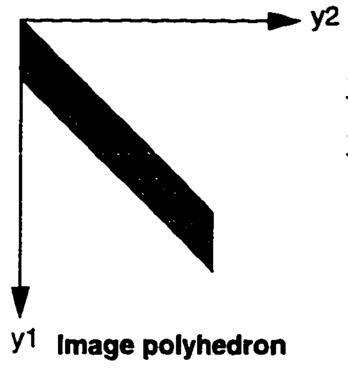
The relaxed bounds correspond to an m-dimensional parallelepiped that encloses the polyhedron representing the transformed array bounds. We call the former the *enclosing parallelepiped* and the latter the *image polyhedron.* As discussed in Section 4.4.1, the relaxed bounds must be in the form (4.7) so that, roughly speaking, they can be linearly transformed to constant bounds without nullifying the benefit of the transformation chosen in Chapter 3. In geometrical terms, we require that the enclosing parallelepiped can be transformed to an m-dimensional rectilinear region; not all parallelepipeds are acceptable.

This section concerns computing the relaxed bounds. Again, Figure 4.4 on page 74 shows how different parts of our discussion fit together. We first elaborate on requirements for the relaxed bounds and, equivalently, the enclosing parallelepiped (Section 4.5.1). Then, our solution is presented. As we shall see, it applies to polyhedra symmetric about a center (Section 4.5.2). These include all parallelepipeds and hence any image polyhedron we may get by transforming the original bounds. The algorithm consists of three steps.

1. Compute the *augmented bounds* — a set of inequalities that is equivalent to the transformed bounds but contains additional information (Section 4.5.3).
2. Compute the center of symmetry of the image polyhedron (Section 4.5.4).
3. Compute the relaxed bounds by selecting a subset from the augmented bounds and supplementing it with other necessary bounds constructed from it (Section 4.5.5).

Although the presentation is mostly in abstract terms for the general m-dimensional case, we refer to the two-dimensional example in Figure 4.6 for concrete illustration whenever appropriate. The transformed bounds and the equivalent image polyhedron are in part (a). They are taken from the right half of Figure 4.1 on page 68. To simplify phrasing and explain concepts in whichever terminology seems most intuitive, we may occasionally refer interchangeably to a set of bounds and its corresponding polyhedron or parallelepiped, or to a vector and its corresponding point.

(a)



$y_1 \geq y_2 \qquad y_1 \leq y_2 + 50$

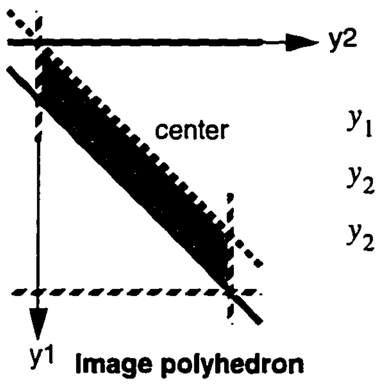
$y_2 \geq 0 \qquad y_2 \leq 1000$

Transformed bounds

$$\begin{bmatrix} 1 & -1 \\ -1 & 1 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \geq \begin{bmatrix} 0 \\ -50 \\ 0 \\ -1000 \end{bmatrix}$$

Run Fourier-Motzkin
+
Compute center

(b)



$y_1 \geq 0 \qquad y_1 \leq 1050$

$y_2 \geq 0 \qquad y_2 \leq 1000$

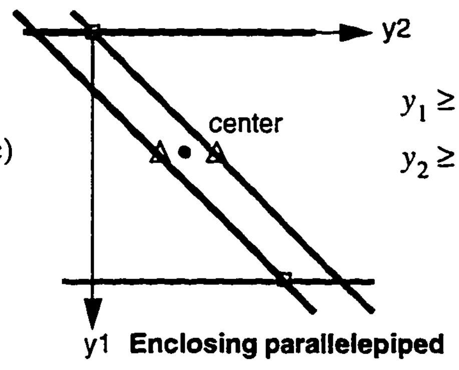
$y_2 \geq y_1 - 50 \qquad y_2 \leq y_1$

Augmented bounds

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \geq \begin{bmatrix} 0 \\ -1050 \\ 0 \\ -1000 \\ -50 \\ 0 \end{bmatrix}$$

Select hyperplanes
+
Build mirror images

(c)



$y_1 \geq 0 \qquad y_1 \leq 1050$

$y_2 \geq y_1 - 50 \qquad y_2 \leq y_1$

Relaxed bounds

$$\begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \geq \begin{bmatrix} 0 \\ -50 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \leq \begin{bmatrix} 1050 \\ 0 \end{bmatrix}$$

Figure 4.6: Relaxing Transformed Array Bounds

4.5.1 Relaxed Bounds Revisited

We have seen in (4.7) that the relaxed bounds must be in the form $r_l \le Ry \le r_u$, where R is lower-triangular with a unit diagonal. Part (c) of Figure 4.6 demonstrates bounds of this form. In scalar terms, the j-th array index, y_j, is bounded by a pair of affine functions in the preceding indices (i.e., $y_1, \ldots, y_{j-1}$) that differ only in the constant term:

$$y_j \ge \sum_{k=1}^{j-1} -R_{jk}y_k + r_{lj} \qquad y_j \le \sum_{k=1}^{j-1} -R_{jk}y_k + r_{uj} \tag{4.19}$$

Geometrically, the enclosing parallelepiped is delineated by m pairs of parallel hyperplanes, one pair for each array dimension. The two hyperplanes for the j-th dimension are defined by (4.19). They must be parallel to the (j+1)-st through m-th dimensions but not the j-th (they may or may not be parallel to the first $j-1$ dimensions). In the case of two dimensions (the first vertical and the second horizontal), this means that we are looking for a parallelogram delineated by a pair of parallel lines ($j = 1$) that are parallel to the second dimension but not the first (in short, horizontal) and another parallel pair ($j = 2$) that are not parallel to the second dimension (in short, vertical or slanted). In other words, we want a parallelogram with a pair of horizontal sides — just as discussed in Section 4.3. In the case of three dimensions (with the third dimension perpendicular to this page of paper), this parallelogram "grows out from the page toward the reader," and the resulting "pillar," whose cross-section is a parallelogram with a pair of horizontal sides, is limited by a third pair of parallel planes ($j = 3$) that are *not* perpendicular to this page (i.e., *not* parallel to the third dimension pointing out from the page).

Furthermore, we want the relaxed bounds to be tight. Specifically, we want to minimize the difference between r_l and r_u in each dimension so that the amount of memory required is minimized. We have seen in (4.16) on page 80 that the restructured array needs memory for roughly $\sum_{k=1}^{m} w_k(r_{uk} - r_{lk}) + 1$ elements. Minimizing $r_{uk} - r_{lk}$ helps to reduce this summation in two ways: it reduces the term $w_k(r_{uk} - r_{lk})$ directly (w_k is the same whatever $r_{uk} - r_{lk}$ might be); it also reduces the terms $w_j(r_{uj} - r_{lj})$ for $1 \le j < k$

indirectly by decreasing the value that w_j must exceed in the algorithm in Figure 4.5 on page 78.

To sum up, the requirements on the relaxed bounds and, equivalently, the enclosing parallelepiped are as follows.

- **Form.** The relaxed bounds and enclosing parallelepiped have the form specified in (4.7), discussed in detail above.
- **Inclusion.** Any vector satisfying the transformed bounds computed in Section 4.1 also satisfies the relaxed bounds. In geometrical terms, the enclosing parallelepiped must, needless to say, enclose the image polyhedron.
- **Tightness.** The relaxed bounds and enclosing parallelepiped are tight. In algebraic terms, $r_{uj} - r_{lj}$ is minimized. In geometrical terms, each pair of parallel faces of the enclosing parallelepiped are as close together as possible.

4.5.2 Symmetric Image Polyhedron

Our algorithm for finding the relaxed bounds applies to convex polyhedra symmetric about a point. By symmetry, we mean that there exists a point c (the center of symmetry or, simply, the center) such that if any point y is in the polyhedron, its mirror image $2c - y$ (the point "on the opposite side of the center") is also in the polyhedron. All three polygons in Figure 4.6 exhibit this symmetry.

Our algorithm always applies to the transformed bounds because the image polyhedron is a parallelepiped (we justify this shortly) and any parallelepiped is symmetric in the sense above. The center of a parallelepiped is the intersection of the hyperplanes exactly halfway between pairs of parallel faces of the parallelepiped. For example, in each polygon in Figure 4.6, the center can be obtained by intersecting the slanted line halfway between the two slanted sides and their horizontal or vertical counterparts.

In the case of transformed array bounds, we can see directly that the image polyhedron is symmetric. We know that the original array indices lie within the original array bounds:

$$b_l \le x \le b_u \tag{4.20}$$

where the vectors b_l and b_u are respectively the lower and upper bounds of the original array. With these, we define

$$c = \frac{1}{2}T(b_l + b_u) \tag{4.21}$$

This must be the center of the image polyhedron. For any point y in the image polyhedron, there is an x satisfying the original bounds (4.20) such that $y = Tx$. The mirror image of y about the center c is

$$2c - y = T(b_l + b_u) - Tx = T(b_l + b_u - x) \tag{4.22}$$

Since x satisfies the original bounds (4.20), so does $b_l + b_u - x$. Therefore, $2c - y$ is also in the image polyhedron.

Having verified that the image polyhedron is symmetric, we discuss in the next few subsections an algorithm to find the relaxed bounds for any polyhedron having this kind of symmetry.

4.5.3 Computing Augmented Bounds

Using the Fourier-Motzkin algorithm [Schrijver 1986], the algorithm first of all computes the augmented bounds — a set of inequalities that is equivalent to the transformed bounds but contains more information. In geometrical terms, the Fourier-Motzkin projects the image polyhedron onto successively lower-dimensional subspaces formed by the first j dimensions (for j from $m - 1$ down to 1). The resulting inequalities are divided into m subsets. The j-th subset delineates the projection of the image polyhedron onto the first j dimensions. Since inequalities in this subset are within the subspace of the first j dimensions, they involve y_j, possibly $y_1, \ldots, y_{j-1}$, but never $y_{j+1}, \ldots, y_m$. In other words, the coefficient for y_j is not zero, those for $y_1, \ldots, y_{j-1}$ may be, and those for $y_{j+1}, \ldots, y_m$ always are. Part (b) of Figure 4.6 illustrates the augmented bounds. The first pair of inequalities, as well as the pair of horizontal lines it corresponds to, delineates the projection of the image polyhedron onto the first dimension y_1. The second and third pairs of inequalities are the transformed bounds, rewritten as lower and upper bounds on y_2 in terms of y_1.

Divided into such subsets, the augmented bounds can be written in the following form. For $1 \le j \le m$, the j-th subset gives the bounds on y_j in terms of the preceding vector components $y_1, \ldots, y_{j-1}$:

$$l_j(y_1, \ldots, y_{j-1}) \le y_j \le u_j(y_1, \ldots, y_{j-1}) \tag{4.23}$$

where $l_j(\ldots)$ ($u_j(\ldots)$) is the lower (upper) bound on y_j and is the maximum (minimum) of one or more affine functions in $y_1, \ldots, y_{j-1}$. (The bounds on y_1 are constants.) For example in Figure 4.6(b), $y_2 \ge 0$ and $y_2 \ge y_1 - 50$ together are written as $y_2 \ge max(0, y_1 - 50)$. Notice that this form of writing the bounds begins to resemble the form required by (4.19) on page 84 in Section 4.5.1. We would get the required form if we could somehow replace $l_j(\ldots)$ and $u_j(\ldots)$ by suitable affine functions in the same arguments, namely $y_1, \ldots, y_{j-1}$. The next few sections aim to do precisely this. In later discussion, we will repeatedly use the following property of the augmented bounds.

Lemma 4.1: For any $y_1, \ldots, y_j$ satisfying the augmented bounds for the first j dimensions, there exists $y_{j+1}, \ldots, y_m$ such that all m components of y satisfy the entire set of augmented bounds.

This property is a known result of the Fourier-Motzkin algorithm [Schrijver 1986]; we do not attempt to justify it on our own. Its converse is trivially true: for any $y_1, \ldots, y_m$ satisfying all the augmented bounds, the first j components of course satisfy the bounds for the first j dimensions. For instance, in part (b) of Figure 4.6, the two thick horizontal lines are augmented bounds for the first dimension. If they were too close together, the augmented bounds would not be equivalent to the transformed bounds because some vectors within the latter would be excluded by the former. If they were too far apart, there would be values of y_1 between them for which there is no y_2 such that the complete vector y lies within the image polyhedron.

In summary, the Fourier-Motzkin algorithm finds the augmented bounds, which have the form (4.23) where the bounds for a dimension are functions of preceding dimensions.

4.5.4 Computing Center of Symmetry

Next, we compute the image polyhedron's center of symmetry. In fact, this can be done directly from the transformed bounds without first computing the augmented bounds, as we have alluded to when arguing in Section 4.5.2 that the image polyhedron is symmetric. We simply apply the index transformation T to the center of the rectilinear region for the original bounds, which is obtained by averaging the lower and upper bounds of each dimension. However, this section presents a more tortuous but also more general method that applies even when the bounds of the restructured array are not simply the image of the original array bounds — a situation arising from partial restructuring, which will be discussed in a later chapter.

With the augmented bounds in (4.23), we can compute the image polyhedron's center of symmetry c recursively as follows:

$$c_j = \frac{1}{2}[l_j(c_1, \ldots, c_{j-1}) + u_j(c_1, \ldots, c_{j-1})] \qquad j = 1, \ldots, m \qquad (4.24)$$

where $l_j(\ldots)$ and $u_j(\ldots)$ are as defined in (4.23). We show below that c is indeed a center of symmetry. In fact, it is *the* center because any center would be equal to c.

Lemma 4.2: Suppose c is defined as in (4.24) and c' is a center of symmetry of the image polyhedron. Then, $c_j = c_j'$ and $l_j(c_1, \ldots, c_{j-1}) \le c_j \le u_j(c_1, \ldots, c_{j-1})$ for all $1 \le j \le m$.

Proof: This can be justified by induction on j. Assume that it is true for all k such that $1 \le k < j$. We argue that it would then be true for j as well. (The case for $j = 1$ can follow exactly the same argument since the assumption is trivially true for $j = 1$.) Our strategy is to show that c_j can be neither greater nor less than c'_j and therefore must equal c'_j.

First, assume for the sake of argument that $c_j > c'_j$. Because $c_1, \ldots, c_{j-1}$ satisfy the bounds for the first $j-1$ dimensions, by Lemma 4.1 there exists a vector γ such that the first $j-1$ components of γ are equal to those of c and γ satisfies all the augmented bounds. In particular,

$$l_j(c_1, \ldots, c_{j-1}) \le \gamma_j \le u_j(c_1, \ldots, c_{j-1}) \tag{4.25}$$

It follows that $u_j(c_1, \ldots, c_{j-1}) \ge l_j(c_1, \ldots, c_{j-1})$. Hence, $c_1, \ldots, c_{j-1}, u_j(c_1, \ldots, c_{j-1})$ satisfy the bounds for the first j dimensions. Again by Lemma 4.1 there exists a point z in the image polyhedron such that $z_1 = c_1, \ldots, z_{j-1} = c_{j-1}$, and $z_j = u_j(c_1, \ldots, c_{j-1})$. Let z' be the mirror image of z about c'. Being the mirror image of z, z' also satisfies the augmented bounds. The j-th component of z' would then be

$$\begin{aligned} z'_j &= 2c'_j - u_j(c_1, \ldots, c_{j-1}) \\ &= 2c_j - u_j(c_1, \ldots, c_{j-1}) + 2(c'_j - c_j) \\ &= l_j(c_1, \ldots, c_{j-1}) + 2(c'_j - c_j) \end{aligned} \tag{4.26}$$

(The last line is obtained by substituting the definition of c_j in (4.24).) This is less than $l_j(c_1, \ldots, c_{j-1})$ because we assume that $c_j > c'_j$. Therefore, z' violates the augmented bounds for the j-th dimension — a contradiction — and the tentative assumption that $c_j > c'_j$ must be false.

The possibility that $c_j < c'_j$ is similarly rejected by considering a point z such that $z_j = l_j(c_1, \ldots, c_{j-1})$ and arguing that the j-th component of its mirror image exceeds $u_j(c_1, \ldots, c_{j-1})$. Hence, the only possibility is that $c_j = c'_j$.

Finally, note that $l_j(c_1, \ldots, c_{j-1}) \le c_j \le u_j(c_1, \ldots, c_{j-1})$ because c_j is the average of $l_j(c_1, \ldots, c_{j-1})$ and $u_j(c_1, \ldots, c_{j-1})$ (see (4.24)), and $u_j(c_1, \ldots, c_{j-1}) \ge l_j(c_1, \ldots, c_{j-1})$ (see (4.25)). This completes the proof.

4.5.5 Computing Relaxed Bounds

Having found the image polyhedron's center of symmetry by (4.24), we are ready to compute the relaxed bounds and, equivalently, the enclosing parallelepiped. As before, we describe the algorithm first and then argue that what it computes meets our requirements.

Algorithm

For j ranging from 1 to m, we evaluate at $c_1, \ldots, c_{j-1}$ the affine functions making up the lower bound function $l_j(\ldots)$ and select the one producing the maximum value. Suppose this affine function has coefficients F_{jk} and offset f_j; in other words, it is given by $\sum_{k=1}^{j-1} F_{jk} y_k + f_j$. We include the following inequality in the relaxed bounds:

$$y_j \ge \sum_{k=1}^{j-1} F_{jk} y_k + f_j \quad \Leftrightarrow \quad y_j - \sum_{k=1}^{j-1} F_{jk} y_k \ge f_j \qquad (4.27)$$

In Figure 4.6(c), these inequalities (one per dimension) are listed on the left and correspond to the two black lines in the index space. They are selected from the augmented bounds in Figure 4.6(b); the striped lines are omitted. We write them in vector form as

$$(I - F)\, y \ge f \qquad (4.28)$$

where F is a lower-triangular matrix with a zero diagonal and whose elements in the lower triangle are F_{jk}, as illustrated by the first vector inequality in Figure 4.6(c). To this set of m inequalities, we add its mirror image about the center:

$$(I - F)\, y \le 2\, (I - F)\, c - f \qquad (4.29)$$

These correspond to the two grey lines. Combining the two sets of inequalities (4.28) and (4.29), we obtain the full set of relaxed bounds:

$$f \le (I-F)\,y \le 2\,(I-F)\,c-f \tag{4.30}$$

These bounds satisfy the three requirements stated in Section 4.5.1: they have the right form, include the transformed bounds, and are tight. We consider each of these in turn.

Form

First, the bounds are in the form required by (4.7) in Section 4.4.1, with

$$r_l = f, \quad r_u = 2\,(I-F)\,c-f, \quad R = I-F \tag{4.31}$$

This follows from the observation that $I-F$ is lower-triangular with a unit diagonal. At this point, we take note of the following lemma on $r_{uj}-r_{lj}$, which is the "distance" between each pair of parallel hyperplanes forming the enclosing parallelepiped. This result will be used in later discussion.

Lemma 4.3: $r_{uj}-r_{lj} = 2c_j - 2l_j(c_1, \dots, c_{j-1})$ for $1 \le j \le m$.

Proof: When computing the relaxed bounds, we include bounds selected from the augmented bounds according to (4.27). Recall that $l_j(y_1, \dots, y_{j-1})$ is the maximum of affine functions in $y_1, \dots, y_{j-1}$, and the algorithm picks the one yielding that maximum at c_1, …, c_{j-1}. Therefore, at those values the maximum and the selected affine function have the same value:

$$l_j(c_1, \dots, c_{j-1}) = \sum_{k=j+1}^{m} F_{jk}c_k + f_j \tag{4.32}$$

Substituting this into the right-hand side of this lemma, we get

$$2c_j - 2l_j(c_1, \dots, c_{j-1}) = 2c_j - 2\left(\sum_{k=j+1}^{m} F_{jk}c_k\right) - 2f_j \tag{4.33}$$

which is the j-th component of

$$2c - 2Fc - 2f = 2(I - F)c - 2f = r_u - r_l \tag{4.34}$$

by the definition of r_l and r_u in (4.31) above. In other words, the right-hand side of this lemma equals $r_{uj} - r_{lj}$, as is to be proved.

Inclusion

Second, any vector y that satisfies the transformed bounds also satisfies the relaxed bounds. To see this, consider each half of the relaxed bounds in turn. For the half selected from the augmented bounds, (4.28), we know that any vector y satisfying the transformed bounds also satisfies all the augmented bounds and, of course, any subset. For the mirror image half, (4.29), note that the mirror image of y about c satisfies the transformed bounds because of the symmetry. Therefore, it also satisfies (4.28), which implies that y satisfies (4.29):

$$(I - F)(2c - y) \geq f \Leftrightarrow (I - F)y \leq 2(I - F)c - f \tag{4.35}$$

Tightness

Third, the relaxed bounds are at least as tight as any set of bounds satisfying the two requirements above. Consider an alternative set of bounds: $r'_l \leq R'y \leq r'_u$. We claim that $(r'_{uj} - r'_{lj}) \geq (r_{uj} - r_{lj})$ for $1 \leq j \leq m$ or, in vector form, $(r'_u - r'_l) \geq (r_u - r_l)$.

To see this, we select two vectors within the transformed bounds that are mirror images of each other about c. For every j, a different pair is chosen and their "distance" along the j-th dimension is considered. In Figure 4.6(c), the points for $j = 1$ are marked by unfilled squares, and those for $j = 2$ by unfilled triangles. In general, the first $j - 1$ components of both vectors are $c_1, \ldots, c_{j-1}$. The other components are chosen as follows. First, from Lemma 4.2 we know that $c_1, \ldots, c_{j-1}$ satisfy the bounds for the first $j - 1$ dimensions. Therefore, by Lemma 4.1 there exists a vector that has the same first

$j-1$ components and satisfies all the bounds, in particular those for the j-th dimension. Its existence implies that the lower bound for the j-th dimension, $l_j(c_1, \ldots, c_{j-1})$, does not exceed the upper bound, $u_j(c_1, \ldots, c_{j-1})$. It follows that $c_1, \ldots, c_{j-1}, l_j(c_1, \ldots, c_{j-1})$ satisfy the bounds for the first j dimensions. Therefore, again by Lemma 4.1, there exists a vector, say z, within the bounds and whose first j components are $c_1, \ldots, c_{j-1}$, and $l_j(c_1, \ldots, c_{j-1})$. Similarly, we construct the vector z' within the bounds and whose first j components are $c_1, \ldots, c_{j-1}$, and $u_j(c_1, \ldots, c_{j-1})$. In summary,

$$\begin{aligned} z_1 &= z'_1 = c_1 \qquad \ldots \qquad z_{j-1} = z'_{j-1} = c_{j-1} \\ z_j &= l_j(c_1, \ldots, c_{j-1}) \\ z'_j &= u_j(c_1, \ldots, c_{j-1}) \end{aligned} \tag{4.36}$$

Since z and z' are within the transformed bounds, they both satisfy the alternative bounds $r'_l \le R'y \le r'_u$. In particular, for the j-th dimension

$$\begin{aligned} r'_{lj} &\le \left[z_j - \sum_{k=j+1}^{m} R'_{jk} z_k \right] = \left[l_j(c_1, \ldots, c_{j-1}) - \sum_{k=j+1}^{m} R'_{jk} c_k \right] \\ r'_{uj} &\ge \left[z'_j - \sum_{k=j+1}^{m} R'_{jk} z'_k \right] = \left[u_j(c_1, \ldots, c_{j-1}) - \sum_{k=j+1}^{m} R'_{jk} c_k \right] \end{aligned} \tag{4.37}$$

Negating the first inequality and adding the second would eliminate the two summations and yield

$$r'_{uj} - r'_{lj} \ge u_j(c_1, \ldots, c_{j-1}) - l_j(c_1, \ldots, c_{j-1}) \tag{4.38}$$

Substituting into the right-hand side the definition of c_j in (4.24) on page 88 and then applying Lemma 4.3 on page 91, we have

$$r'_{uj} - r'_{lj} \ge 2\,c_j - 2\,l_j(c_1, \ldots, c_{j-1}) = r_{uj} - r_{lj} \tag{4.39}$$

as we wish to prove.

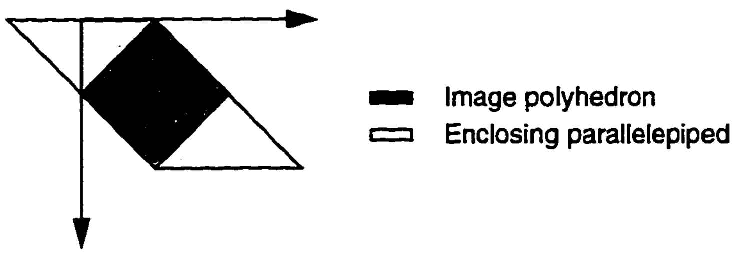


Figure 4.7: Worst-Case Example of Memory Use

In conclusion, the relaxed bounds as given by (4.30) are at least as tight as alternative bounds — in every dimension. They are therefore as efficient in memory use as other bounds of the same form.

Using the algorithm presented so far in this chapter, we allocate for the restructured array at most roughly $m!$ times the memory needed for storing array elements, where m is the number of array dimensions. A proof is given in Appendix A. For two dimensions, the worst case occurs when, for example, the image polyhedron is a square with horizontal and vertical diagonals, as Figure 4.7 illustrates. A worst case in three dimensions is analogous: a cube with diagonals along the three natural, orthogonal dimensions. Such cases are rare in our experience. Access patterns that require skewing of array indices typically occur in banded matrix applications, where the arrays have one long dimension and one or more short dimensions to store a narrow band of matrix elements.

4.5.6 Summary

Our algorithm for computing the relaxed bounds applies to transformed bounds that are symmetric. First, using the Fourier-Motzkin algorithm, the algorithm computes the augmented bounds, which are equivalent to the transformed bounds but divided into bounds for each dimension dependent on preceding dimensions only. Next, we compute the center of symmetry according to (4.24). Finally, the relaxed bounds are constructed from a specially chosen subset of the augmented bounds and its mirror image according to (4.30).

The relaxed bounds thus computed are shown to have the required form, to include all vectors included by the transformed bounds, and to require the least memory among bounds meeting the other two conditions.

4.6 Summary

Laying out elements of the restructured array is difficult. The traditional method does not apply because of the generality of our index transformations. We have to find a linearization vector for the restructured array that is integral and represents a row-major storage order — the key assumption in our earlier analysis. To do that, we first compute the bounds of the restructured array by linearly transforming the original array bounds. Then, the transformed bounds are relaxed to a special form amenable to a further transformation that preserves the improvement of previous transformations. Finally, we calculate a linearization vector from the relaxed bounds.

Chapter 5

Partial Restructuring

In the preceding chapters, we have assumed that restructuring is always applied to the entire array. However, a loop may access only a small part of an array, in which case restructuring the whole array is unnecessary. For example, a loop may use only part of a large array that is statically declared to accommodate the maximum problem size. Sometimes, an array access may cover only a subspace of the full array index space, such as a row, column, or diagonal of a two-dimensional array. Also, an access may touch only a subset of regularly spaced elements in an array, such as the elements with even indices.

In this chapter, we discuss restructuring only that part of an array that is accessed by a given loop. In other words, only the accessed elements are placed in the restructured array. Not only does this reduce memory and copying overheads by decreasing the size of the restructured array, it also improves spatial locality further because array elements are stored more compactly in fewer cache lines. We describe how to determine the set of accessed elements and how to extract only those elements into the restructured array — without deviating from the affine framework of Chapter 2. In those cases that our partial restructuring techniques do not handle, we can and do fall back to restructuring the entire array.

Two sets of techniques are presented. The first deals with elements regularly spaced in the index space (like a checkerboard, for instance). The second applies when the loop touches elements in a restricted region of the index space (e.g., only those with indices between 1 and 100).

5.1 Regularly Spaced Elements

This section presents techniques to restructure only elements that are regularly spaced in the index space in the form of a "lattice." Such access patterns arise from array indices with non-unit strides. We start by extending the transformation framework. This is followed by an algorithm to choose the transformation for a single access and finally techniques for multiple accesses. As in previous chapters, array bounds are not considered for the moment; they are dealt with in the next section.

5.1.1 Extending the Framework

Consider the example in Figure 5.1. We concentrate on the upper half here; the lower half is discussed later in Section 5.1.2. Both the inner and outer loops go through the two-dimensional array at strides of two. Only elements with an even row index (x_1) and odd column index (x_2) are accessed. To improve locality, not only should we transpose the array so that the inner loop accesses a row of elements, but we also want the restructured array to contain only those accessed elements. The figure shows how this is achieved.

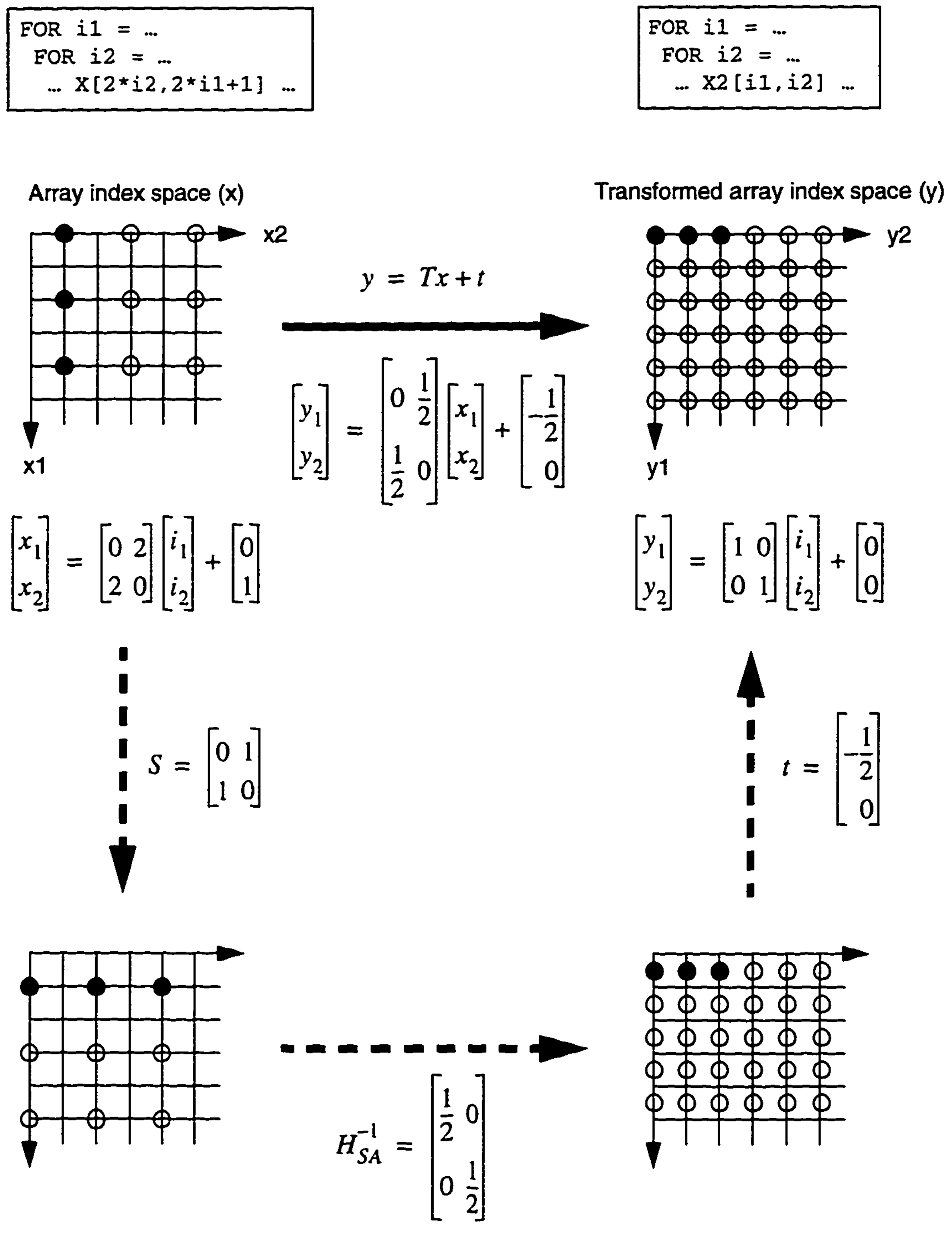
The index transformation is extended from $y = Tx$ to

$$y = Tx + t \tag{5.1}$$

where x and y, as before, are the original and transformed index vectors respectively, T is an $m \times m$ transformation matrix, and t an m-dimensional offset vector (m being the number of array dimensions). This transformation departs from earlier discussion in two ways:

- its matrix T is nonsingular — possibly non-integral and hence non-unimodular;
- it has an additional offset vector t, which may also be non-integral.

Because of the non-integral matrix T and offset vector t, some (integral) original index vectors x may be mapped to non-integral vectors in the transformed index space. Being non-integral, the latter cannot identify any element in the restructured array. However, this



```
FOR i1 = ...
 FOR i2 = ...
  ... X[2*i2,2*i1+1] ...
```

```
FOR i1 = ...
 FOR i2 = ...
  ... X2[i1,i2] ...
```

$$y = Tx + t$$

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 0 & \frac{1}{2} \\ \frac{1}{2} & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} -\frac{1}{2} \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 & 2 \\ 2 & 0 \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$S = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

$$t = \begin{bmatrix} -\frac{1}{2} \\ 0 \end{bmatrix}$$

$$H_{SA}^{-1} = \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix}$$

Figure 5.1: Restructuring Regularly Spaced Elements

is not a problem. Such original index vectors correspond to elements that are not accessed anyway; there are no corresponding elements in the restructured array, as it should be. Index vectors for elements that *are* accessed are always mapped to integral vectors, which *do* identify the corresponding restructured array elements. In fact, this is precisely the mechanism by which unaccessed elements (whose transformed index vectors would be non-integral) are omitted from the restructured array. We say that an element is *selected* by the index transformation if its transformed index vector is integral, and *omitted* otherwise.

Although T or t may be non-integral, index computation can still be done strictly in integer arithmetic since in choosing T and t we ensure that the transformed access has an integral access matrix and offset vector. This is crucial because it means that the usual compiler code generation and optimization techniques continue to apply despite the non-integral index transformation.

The extra offset vector t does not affect the access pattern, but merely shifts it. This neither improves nor degrades locality. Recall that in Section 2.3 we have considered adding an offset vector to our linear array transformation but have decided against it because it does not change the storage order of elements. That conclusion remains true. We add the offset here only to make the transformed array indices integral.

Having introduced the basic idea of restructuring regularly spaced elements, we next consider how the transformation is chosen.

5.1.2 A Single Access

As mentioned above, we extend the index transformation from linear to affine and the transformation matrix from unimodular to nonsingular: $y = Tx + t$. Neither T nor t is necessarily integral, although x and y must be. Since the transformed index vector y is an affine function of the original index vector x, which is itself an affine function of the iteration vector i (i.e., $x = Ai + a$) we can express y directly in terms of i:

$$y = Tx + t = (TA)\,i + (Ta + t) \qquad (5.2)$$

The use of affine rather than linear index transformations complicates the transformed array bounds only slightly. Instead of $x = T^{-1}y$, we have $x = T^{-1}(y-t)$. Hence, the original array bounds, written in the form of a vector inequality $Bx \geq b$ (see (4.4) on page 67), are transformed to $\left(BT^{-1}\right)y \geq \left(b + BT^{-1}t\right)$ (c.f. (4.6) on page 68). The rest of the linearization procedure in Chapter 4 still applies.

This section discusses what we require of T and t and how to compute them. Only a single array access is considered; the case for multiple accesses is left to the next section.

Requirements on *T* and *t*

Our requirement for the transformation consists of two parts that mirror each other. First, any iteration vector i, which is of course integral, must be mapped to an integral transformed index vector y. In other words, any element accessed by some iteration must have an integral transformed index vector and therefore is selected by the index transformation. Related to this, we further require that y can be computed from i using only integer arithmetic. Second, and conversely, any integral transformed index vector y should correspond to one or more integral iteration vectors i. This means that all elements in the restructured array really are accessed by some iterations or, equivalently, all elements selected by the index transformation are accessed by some iteration(s). For example in Figure 5.1, if the transformation were only an array transpose, the element `X2[3,1]` would store the value of `X[1,3]`, which is not accessed by any iteration because its row index is odd.

To satisfy the two requirements above, we require that

- TA and $Ta + t$ are both integral, and
- TA's Hermite normal form[1] is the identity matrix.

1. For any $m \times n$ matrix of full row rank, say Γ, there exists an $n \times n$ unimodular matrix U such that $\Gamma U = [H_\Gamma \;\; 0]$ where the $m \times m$ matrix H_Γ is in Hermite normal form (see page 44 for an earlier discussion on Hermite normal form). H_Γ is the Hermite normal form of Γ [Schrijver 1986].

Let us consider each in turn. First, in order that any integral i is mapped to an integral y according to (5.2), we want both TA and $Ta + t$ to be integral. This is clearly a sufficient condition, guaranteeing that y can be computed from i using only integer operations. It is also a necessary condition. If TA were not integral (say in the j-th column), two integral vectors differing only in the j-th dimension by 1 would be mapped to two vectors differing by the j-th column of TA, at least one of which must be non-integral because their difference is not integral. Thus, TA has to be integral. Given that, $Ta + t$ must also be integral because y, i, TA are all integral, and $y = (TA)\,i + (Ta + t)$.

Second, in order that any integral y corresponds to some integral i according to (5.2), we want the Hermite normal form of TA, denoted H_{TA}, to be the identity matrix. We now explain why. If H_{TA} equals the identity matrix, then by the definition of Hermite normal form, $(TA)\,U = [I \;\; 0]$ for some $n \times n$ unimodular matrix U. In other words,

$$TA = [I \;\; 0]\,U^{-1} \tag{5.3}$$

Thus, given any integral vector y, we can find a corresponding iteration vector i (there may be many) as follows: lengthen the m-dimensional vector $y - (Ta + t)$ to n dimensions with $n - m$ appended zeros, and left-multiply the lengthened vector by U. To verify that the resulting i corresponds to the given y, we substitute it into the right-hand side of (5.2):

$$\begin{aligned} (TA)\,i + (Ta + t) &= \left([I \;\; 0]\,U^{-1}\right)\left(U\begin{bmatrix} y - (Ta + t) \\ 0 \end{bmatrix}\right) + (Ta + t) \\ &= [I \;\; 0]\begin{bmatrix} y - (Ta + t) \\ 0 \end{bmatrix} + (Ta + t) \\ &= y \end{aligned} \tag{5.4}$$

Finding T and t

We now describe how to compute T and t to meet the above requirements: TA and $Ta + t$ are integral, and TA 's Hermite normal form is the identity matrix. This is done in three steps as illustrated in the lower half of Figure 5.1.

1. Compute a nonsingular matrix, denoted S, to flatten the columns of the access matrix exactly as described earlier in Section 3.2.2. Roughly speaking, this matrix represents a transformation to reorder the elements as appropriate. In Figure 5.1, S is $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, corresponding to an array transpose to make the innermost loop go through array elements row by row.

2. Compute the Hermite normal form[2] of SA, denoted H_{SA}. Choose $T = H_{SA}^{-1} S$. Intuitively, the matrix H_{SA}^{-1} represents an additional transformation to omit unaccessed elements while preserving the effect of the previous transformation S. In the example, array indices are halved. As Figure 5.1 illustrates, this may lead to non-integral array indices. (Notice that the circles lie between grid points.) This problem is remedied by the next step.

3. Choose $t = -Ta$. Adding the offset vectors t makes the non-integral index vectors integral by shifting all of them by a suitable non-integral amount (in the example, up by $\frac{1}{2}$).

We now argue more formally that these choices for T and t meet our requirements. First of all, choosing $-Ta$ for t makes $Ta + t$ zero and hence integral. As for TA, by the definition of Hermite normal form $(SA)\,U = [H_{SA} \quad 0]$ for some unimodular matrix U. Therefore,

$$TA = \left(H_{SA}^{-1} S\right) A = H_{SA}^{-1}(SA) = H_{SA}^{-1}[H_{SA} \quad 0]\,U^{-1} = [I \quad 0]\,U^{-1} \qquad (5.5)$$

Because U^{-1} is the inverse of a unimodular matrix, it is also unimodular and thus by integral by definition. Furthermore, this very equation also indicates that the Hermite normal form of TA is I, as required. Thus, the requirements on TA are also met.

2. Strictly speaking, the Hermite normal form exists only for matrices of full row rank (i.e., the rank is equal to the number of rows). The access matrix A is not necessarily of full row rank; hence neither is SA. However, our algorithm for finding S from A (Figure 3.6 on page 39) ensures that the height of matrix SA is the same as its rank. (Recall that our algorithm guarantees that the transformed access matrix has identical height and rank profiles.) Therefore, SA consists of $m - rank(SA)$ rows of zeros followed by a $rank(SA) \times n$ submatrix of full row rank. In the rest of our discussion, "SA" refers to this submatrix.

Finally, TA and SA have identical height profiles because as $TA = H_{SA}^{-1}(SA)$ and H_{SA}^{-1}, like its inverse H_{SA}, is lower-triangular. This means that T and S are equally "good" in flattening the columns of the access matrix.

5.1.3 Multiple Accesses

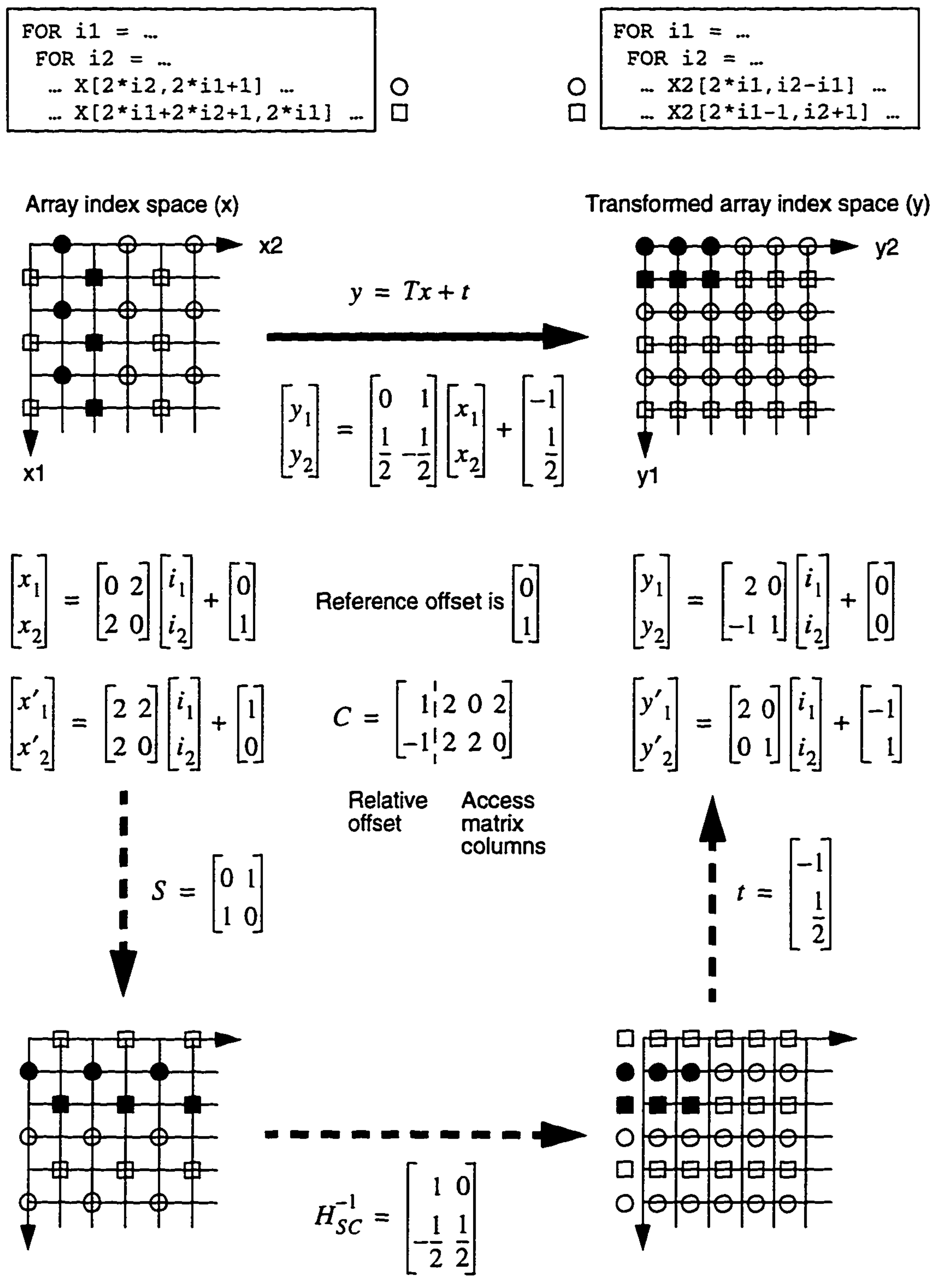
Multiple accesses are handled with a slightly adapted algorithm. As usual, we start with our requirements and then present the algorithm for satisfying them. Figure 5.2 illustrates the algorithm. In this example, the computed transformation transposes the array so that the inner loop goes through a row of elements, and omits from the restructured array elements that are not accessed at all.

Requirements

The requirements are similar to those in the last section. In order that the restructured array includes all accessed elements, for every access, any integral iteration vector i must be mapped to an integral transformed index vector y. As before, we ensure this by requiring that both TA and $Ta + t$ are integral for each access. However, it may not always be possible to satisfy the converse requirement because of the regularity of affine index transformations. In other words, not every integral y corresponds to some integral iteration vector i for at least one of the accesses; the restructured array could contain elements not really accessed by any iteration. Nevertheless, the index transformation we compute is minimal: it selects no more elements than any affine transformation satisfying the first requirement. Before justifying this claim, let us first describe how we compute the transformation.

Algorithm

The algorithm to compute T and t resembles that in the last section for a single access. In fact, it reduces to the latter when there is only one access. The main difference lies in



```
FOR i1 = ...
 FOR i2 = ...
  ... X[2*i2,2*i1+1] ...           ○
  ... X[2*i1+2*i2+1,2*i1] ...      □
```

```
  ○  FOR i1 = ...
  □   FOR i2 = ...
       ... X2[2*i1,i2-i1] ...
       ... X2[2*i1-1,i2+1] ...
```

$$y = Tx + t$$

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ \frac{1}{2} & -\frac{1}{2} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} -1 \\ \frac{1}{2} \end{bmatrix}$$

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 & 2 \\ 2 & 0 \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \qquad \text{Reference offset is} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \qquad \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} x'_1 \\ x'_2 \end{bmatrix} = \begin{bmatrix} 2 & 2 \\ 2 & 0 \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad C = \left[\begin{array}{c|ccc} 1 & 2 & 0 & 2 \\ -1 & 2 & 2 & 0 \end{array}\right] \qquad \begin{bmatrix} y'_1 \\ y'_2 \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} + \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

$$S = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \qquad t = \begin{bmatrix} -1 \\ \frac{1}{2} \end{bmatrix}$$

$$H_{SC}^{-1} = \begin{bmatrix} 1 & 0 \\ -\frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

Figure 5.2: Restructuring Regularly Spaced Elements for Multiple Accesses

the new steps 2 and 3 to construct a special matrix. The lower half of Figure 5.2 serves as our example in the following discussion.

1. Compute matrix S from the aggregate matrix formed from columns of the individual access matrices, as described before in Section 3.3.1. This transformation reorders elements for better locality. In the figure, the matrix $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ in effect transposes the array. As in the last section, what remains to be done is finding further transformations to omit the unaccessed elements without nullifying the effect of S.

2. Choose any one of the accesses as the "reference." Call the offset vector of the reference access the *reference offset*, denoted a_{ref}. In Figure 5.2, we pick the first access as the reference; a_{ref} is thus $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$. For each access, compute its *relative offset*, namely the difference between its offset vector and the reference offset. The array index vector can thus be expressed as $Ai + (a - a_{ref}) + a_{ref}$. Below, we will make each access's $Ta + t$ integral by making both $T(a - a_{ref})$ and $Ta_{ref} + t$ integral.

3. Compose a matrix, C, with relative offsets and access matrix columns from all accesses. (In Figure 5.2, C is shown in the middle with a dashed line separating the two parts.) The columns of C can be in any order. We will explain the significance of C shortly. Note that when there is only one access, C is simply the access matrix, and the next two steps reduce to steps 2 and 3 in the last section.

4. Compute the Hermite normal form of SC, denoted H_{SC}^{-1}. Choose $T = H_{SC}^{-1} S$. As before, H_{SC}^{-1} intuitively represents a transformation that omits unaccessed elements from the restructured array.

5. Choose $t = -Ta_{ref}$, like in the last section, to shift index vectors resulting from step 4 so that they all become integral.

As in the last section, T is as good as S in flattening access matrix columns. Because $T = H_{SC}^{-1} S$ and H_{SC}^{-1} is lower-triangular, for any access matrix A both transformed access matrices SA and $TA = H_{SC}^{-1}(SA)$ must have the same height profile.

Moreover, the index transformation thus computed ensures that for every access, TA and $Ta + t$ are integral as required. Let us explain why. First of all, because H_{SC} is the Hermite normal form of SC, TC is integral, for the same reason as why TA is integral in the last section. Recall that the columns of C are each access's relative offset $a - a_{ref}$ and access matrix columns. Therefore, the product of T with them must be integral: for every access, $T(a - a_{ref})$ and TA are integral. Moreover, step 5 ensures that $Ta_{ref} + t$ is integral (in fact, zero). It follows from these two observations that for each access, TA and $Ta + t$ (which equals $T(a - a_{ref}) + (Ta_{ref} + t)$) are both integral, as we require.

Computed Index Transformation Is Minimal

Although the computed index transformation may select elements that are not really accessed, it is minimal in that it selects no more elements than other index transformations that also select all the accessed elements. In other words, any element selected by our transformation is also selected by such an alternative. Let the alternative index transformation be $y = T'x + t'$.

To justify this claim, we first have to argue that $T'C$ must be integral. For the sake of argument sake, suppose it were not. Recall that the columns of C originate from access matrices or relative offsets. Again for the sake of argument, suppose further that one of the non-integral columns in $T'C$ originates from the j-th column of some access matrix A. Then, for that access, the transformed access function $y = (T'A)i + (T'a + t')$ would map two iteration vectors differing by 1 in the j-th dimension to two transformed index vectors differing by the j-th column of $T'A$. If, as we have assumed, this column is not integral, the two index vectors cannot both be integral and hence the two elements cannot both be selected by the alternative transformation, even though both elements are accessed and therefore should be selected. Hence, the non-integral columns in $T'C$, if any, could not have originated from access matrices.

Moreover, they could not have originated from relative offsets either. Again, suppose for argument's sake that one of them were. Let that access be $Ai + a$. The transformed access $(T'A)\,i + (T'a + t')$ can be written as

$$(T'A)\,i + T'(a - a_{ref}) + (T'a_{ref} + t')\,. \tag{5.6}$$

Compare it with the transformed reference access:

$$(T'A_{ref})\,i + (T'a_{ref} + t') \tag{5.7}$$

Note that $T'A$ and $T'A_{ref}$ are integral as we have concluded in the last paragraph, but $T'(a - a_{ref})$ is non-integral as we assume in this paragraph. Therefore, for any integral iteration vector *i*, the two index vectors (5.6) and (5.7) cannot both be integral: if (5.6) is integral, $(T'a_{ref} + t')$ is not and hence neither is (5.7); if (5.7) is integral, $(T'a_{ref} + t')$ is also integral and hence (5.6) must be non-integral. In other words, these two elements accessed by iteration *i* cannot both be selected even though they should. Again, we reach a contradiction and therefore must conclude that $T'C$ is integral.

Having shown that $T'C$ is integral, we can now argue that any element selected by our index transformation (i.e., $Tx + t$ is integral) is selected by the alternative as well (i.e., $T'x + t'$ is also integral). With some minor algebraic manipulations, we have

$$T'x + t' = \left(T'T^{-1}\right)(Tx + t) + \left(t' - T'T^{-1}t\right) \tag{5.8}$$

Based on this, an integral $Tx + t$ would always imply an integral $T'x + t'$ if both $T'T^{-1}$ and $t' - T'T^{-1}t$ are integral, which indeed they are as we show below.

First, let us consider $T'T^{-1}$. Substituting the choice for T in step 4 above, we have

$$T'C = T'T^{-1}TC = \left(T'T^{-1}\right)\left(H_{SC}^{-1}\,S\right)C \tag{5.9}$$

By the definition of Hermite normal form, there exists a unimodular matrix U such that $SC = [H_{SC} \;\; 0]\,U$. Substituting this into (5.9) gives us

$$T'C = \left(T'T^{-1}\right)H_{SC}^{-1}[H_{SC} \;\; 0]\,U = \left(T'T^{-1}\right)[I \;\; 0]\,U = \left[T'T^{-1} \;\; 0\right]U \quad (5.10)$$

which implies that

$$(T'C)\,U^{-1} = \left[T'T^{-1} \;\; 0\right] \quad (5.11)$$

Since U is unimodular, so is U^{-1} [Schrijver 1986]. By the definition of a unimodular matrix, U^{-1} is integral. Moreover, we have shown above that $T'C$ is integral. Therefore, $T'T^{-1}$ is also integral.

Next, we consider $t' - T'T^{-1}t$. Let x be the original index vector of some element accessed by the loop nest. Both our index transformation and the alternative must select this element. In other words, both $Tx + t$ and $\left(T'T^{-1}\right)(Tx + t) + \left(t' - T'T^{-1}t\right)$ (see (5.8)) are integral. As we already know that $T'T^{-1}$ is integral and hence so is the first term on the right-hand side, $t' - T'T^{-1}t$ must be integral also. As $T'T^{-1}$ and $t' - T'T^{-1}t$ are both integral, according to (5.8) $T'x + t'$ is integral whenever $Tx + t$. This completes the argument for our claim that any element selected by the computed index transformation is also selected by the alternative transformation.

5.1.4 Summary

We have discussed techniques to restructure a subset of elements regularly spaced in the index space. The index transformation is affine and has a nonsingular, possibly non-unimodular or even non-integral, transformation matrix. Although the transformation may be non-integral, the algorithm to find the transformation matrix T and offset vector t guarantees that all the transformed access matrices and offsets, namely TA and $Ta + t$, are integral. This ensures that the restructured array contains all the elements accessed by the loop but omits those that are not, to the extent allowed by affine index transformations. Specifically, when there is only one access, we further ensure that our index transformation selects only elements that are actually accessed, and no more. When there are multiple accesses, however, we cannot guarantee this but our index transformation is minimal in that it selects no more elements than alternative transformations.

5.2 Range of Index Vectors

The restructured array need not always contain all the elements of the original array. In some cases, a loop may access only elements with indices within a restricted range, for example elements 1 to 100 among the 200 elements along some array dimension or one column of a two-dimensional array. By using such restricted ranges in place of the original array bounds for our linearization procedure, we can include in the restructured array only those elements that are really accessed.

This section discusses how to compute, though sometimes approximately, the range of accessed elements from array index expressions and loop bounds. The required assumptions will be presented as they become relevant to the discussion. The analysis below is not critical for array restructuring: whenever a necessary assumption does not hold, we can (and do) always fall back to the option of restructuring entire arrays based on transformed array bounds.

We first explain the problem. Then, we present solutions for a single access, followed by a solution in the case of multiple accesses. We close this section by discussing the how partial restructuring affects the linearization procedure presented earlier.

5.2.1 The Problem

After choosing an index transformation as in Section 5.1, we want to find the "image" of the loop bounds in the transformed index space. Specifically, we wish to compute the set of transformed index vectors that correspond to one or more iteration vectors *within the loop bounds*.

Even for a single affine access, this problem is much harder than the seemingly similar problem solved in Section 4.1 — computing the transformed array bounds from the original array bounds. In that problem, the transformation is invertible because the transformation matrix T is nonsingular. Here, however, the transformed access matrix TA may be singular, and therefore the mapping may not be invertible.

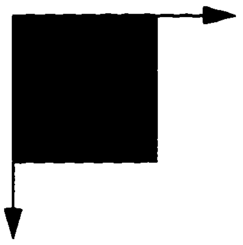
Figure 5.3 illustrates the difficulties. Part (a) shows the easy case where TA is nonsingular. The image of the loop bounds can be computed just as before by substituting $i = (TA)^{-1}[y - (Ta + t)]$ into the bounds on i (i.e., the loop bounds). In part (b), TA is of full column rank but not full row rank. The above substitution is invalid because TA has no inverse. The single loop accesses only one dimension, specifically a row, of a two-dimensional array. Part (c) is difficult for the opposite reason: TA is of full row rank but not full column rank. The two inner loops covers a square in the index space; the outermost loop shifts this square diagonally, sweeping out the shaded region. In both parts (b) and (c), it is not obvious how to find the image. Finally, part (d) combines the difficulties of (b) and (c): TA is neither of full row rank nor of full column rank. The inner loop covers a diagonal line segment; the outer loop shifts it diagonally, thus sweeping out a longer diagonal line segment. While it is perhaps possible to handle each special case with a specialized technique, we prefer to deal with them uniformly, as we now discuss.

5.2.2 Solutions

The problem is to find the image of the loop bounds in the transformed index space. We make two assumptions. First, the array access has affine index expressions. Second, the loop bounds can be represented as a conjunctive set of linear inequalities on the loop variables or, geometrically, as a convex polyhedron in the iteration space. In other words, each lower (upper) loop bound is the maximum (minimum) of one or more affine functions of enclosing loop variables. As mentioned earlier, we fall back to restructuring the entire array whenever our assumptions do not hold.

```
FOR i1 = 0, 100
 FOR i2 = 0, 100
  ... X2[i1,i2] ...
```

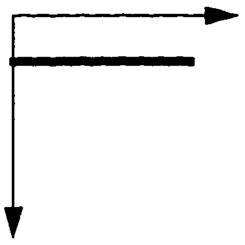
$$TA = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \text{(rank = 2)}$$



(a) Full row and column rank

```
FOR i1 = 0, 100
 ... X2[3,i1] ...
```

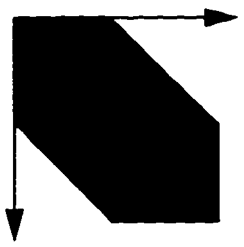
$$TA = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad \text{(rank = 1)}$$



(b) Full column rank only

```
FOR i1 = 0, 100
 FOR i2 = 0, 100
  FOR i3 = 0, 100
   ... X2[i1+i2,i1+i3] ...
```

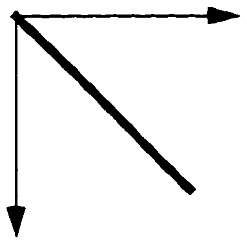
$$TA = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \quad \text{(rank = 2)}$$



(c) Full row rank only

```
FOR i1 = 0, 100
 FOR i2 = 0, 100
  ... X2[i1+i2,i1+i2] ...
```

$$TA = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \quad \text{(rank = 1)}$$



(d) Neither full column rank nor full row rank

Figure 5.3: Examples of Restricted Index Ranges

With these assumptions, we can refine the problem statement as follows: given a transformed array access with access function $Fi+f$, compute the set of index vectors y such that $y = Fi+f$ for one or more iteration vectors i satisfying the loop bounds. The result is to be represented as a set of linear inequalities on the transformed array indices or, geometrically, as a convex polyhedron in the transformed index space. We discuss two ways to calculate this result. They both compute the required set of index vectors, although they are not guaranteed to produce identical sets of inequalities. The first way is conceptually simpler, but the second has been implemented for efficiency.

One way is to project a polyhedron in the $(m+n)$-dimensional vector space of $\begin{bmatrix} y \\ i \end{bmatrix}$ onto the m-dimensional subspace of y alone[3]. Specifically, a vector in the $(m+n)$-dimensional vector space is $\begin{bmatrix} y \\ i \end{bmatrix}$, the vector formed by concatenating y and i. Define a polyhedron, P, in this $(m+n)$-dimensional vector space with the following inequalities:

$$\begin{gathered} Di \geq d \\ y \geq Fi+f \qquad y \leq Fi+f \end{gathered} \tag{5.12}$$

The first line is the loop bounds; the second simply means $y = Fi+f$. Projecting P onto the m-dimensional subspace of y produces the set of y for which there is some i such that y and i together satisfy all the inequalities defining P, that is $y = Fi+f$ with i within the loop bounds. Therefore, this set of y is precisely what we try to compute: the image of the loop bounds in the transformed index space. To find the projection, we can apply the Fourier-Motzkin algorithm to P to get bounds for each dimension of $\begin{bmatrix} y \\ i \end{bmatrix}$ in terms of preceding dimensions of the vector, and retain only the bounds for dimensions corresponding to y. By Lemma 4.1 on page 87, this procedure yields the aforesaid set of y.

While this method is conceptually simple, we have not implemented it mainly because of concerns for efficiency. The Fourier-Motzkin algorithm sometimes produces redundant

3. Amarasinghe and Lam have used a similar approach involving the projection of polyhedra to tackle another problem: generating communication code for distributed-memory multiprocessors [Amarasinghe and Lam 1993].

inequalities. In particular, writing equations as pairs of inequalities may contribute to this problem [Amarasinghe and Lam 1993]. To avoid generating a large number of spurious inequalities, we use instead an algorithm that treats equations as equations.

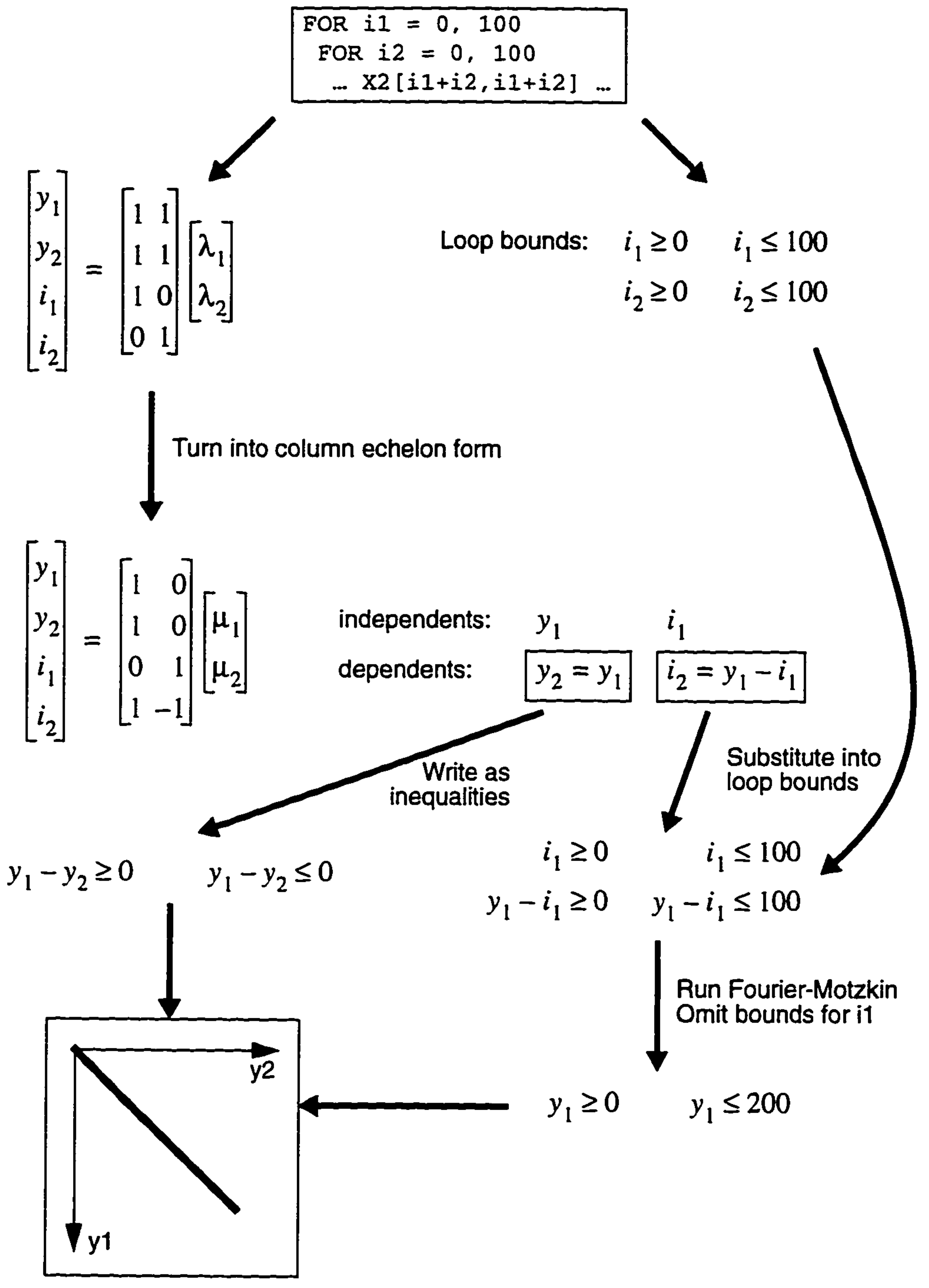
In the second method, roughly speaking we express i as a function of y ("inverting" the function $Fi+f$) and then substitute it into the loop bounds to get equivalent bounds on the index vector y. We have seen that if F is nonsingular, i can be written in terms of y as $i = F^{-1}(y-f)$. If F is singular, however, this fails because F has no inverse. Instead, we express i in terms of *some* components of y and possibly some other parameters (both of which we call *independent components*, or simply *independents*), substitute that expression into the loop bounds to obtain equivalent bounds on the independents, and finally project the latter bounds onto the subspace spanned by the independents in y. We now explain this in more detail with the help of Figure 5.4, which shows how the example in Figure 5.3(d) is handled step by step.

First of all, to express i in terms of the independents (to "invert" $Fi+f$), we treat $y = Fi+f$ as an equation in unknowns y and i, rather than a formula to compute y from i. The "solutions" to this equation can be expressed parametrically as

$$\begin{bmatrix} y \\ i \end{bmatrix} = \begin{bmatrix} F \\ I \end{bmatrix} \lambda + \begin{bmatrix} f \\ 0 \end{bmatrix} \tag{5.13}$$

where λ is an arbitrary n-dimensional vector. Any λ would give us y and i such that $y = Fi+f$ because, in fact, the parametric vector λ is equal to i.

We want to express the same solutions in another way such that the parametric vector resembles y more closely. To do this, we first convert $\begin{bmatrix} F \\ I \end{bmatrix}$ into *column echelon form* [Bloom 1979]. (The implications are discussed shortly.) For example, the matrix in the middle of Figure 5.4 is in this form. Intuitively, the top nonzero in each column descends like a staircase as we go through the columns from left to right, is the only nonzero in the row, and equals 1. Formally, a matrix is in column echelon form if and only if



```
FOR i1 = 0, 100
 FOR i2 = 0, 100
  … X2[i1+i2,i1+i2] …
```

$\begin{bmatrix} y_1 \\ y_2 \\ i_1 \\ i_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \end{bmatrix}$
Loop bounds: $i_1 \geq 0$ $i_1 \leq 100$
$i_2 \geq 0$ $i_2 \leq 100$
Turn into column echelon form
$\begin{bmatrix} y_1 \\ y_2 \\ i_1 \\ i_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}$
independents: y_1 i_1
dependents: $\boxed{y_2 = y_1}$ $\boxed{i_2 = y_1 - i_1}$
Write as inequalities
Substitute into loop bounds
$y_1 - y_2 \geq 0$ $y_1 - y_2 \leq 0$
$i_1 \geq 0$ $i_1 \leq 100$
$y_1 - i_1 \geq 0$ $y_1 - i_1 \leq 100$
Run Fourier-Motzkin
Omit bounds for i1
$y_1 \geq 0$ $y_1 \leq 200$
y2
y1
Resulting bounds on y

Figure 5.4: Computing Image of Loop Bounds

- it consists of some (or no) nonzero columns followed by some (or no) zero columns;
- in each nonzero column, if any, the top nonzero
  - equals 1,
  - is the only nonzero in its row, and
  - occurs below the top nonzero of the column to the left, if any.

The matrix $\begin{bmatrix} F \\ I \end{bmatrix}$ (or any matrix for that matter) can be converted to this form by elementary column operations, whose effects can be summarized in a nonsingular matrix multiplied to $\begin{bmatrix} F \\ I \end{bmatrix}$ on the right [Bloom 1979]. That is, there is a nonsingular matrix Δ such that $\begin{bmatrix} F \\ I \end{bmatrix}\Delta$, denoted E, is in column echelon form. Thus, the solutions to the "equation" $y = Fi + f$ can also be expressed as

$$\begin{bmatrix} y \\ i \end{bmatrix} = E\mu + \begin{bmatrix} f \\ 0 \end{bmatrix} \qquad (5.14)$$

where μ, like λ, is an arbitrary n-dimensional vector. This is equivalent to (5.13) because any solution given by a certain λ can also be given by some μ ($\mu = \Delta^{-1}\lambda$), and vice versa ($\lambda = \Delta\mu$).

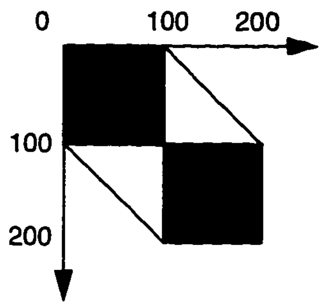
The fact that E is in column echelon form allows i to be expressed in part as a function of y. Specifically, the parametric vector μ in effect consists of some, possibly all, components of index vector y and perhaps some components of iteration vector i — both identified by rows of E that contain the top nonzeros of the columns and therefore, the column echelon form specifies, contain a single "1." (There must be a unique row for each column since column echelon form also specifies that the top nonzero descends through the columns.) In Figure 5.4, for instance, the first and third rows fit the above description. They mean that $\mu_1 = y_1$ and $\mu_2 = i_1$. Thus, y_1 and i_1 are the independent components. The other components of y and i (the "dependents" y_2 and i_2) are expressed in terms of the independents y_1 and i_1, as the diagram illustrates.

After i has been expressed in terms of the independents, it can be substituted into the loop bounds to yield equivalent bounds on the independents. The latter are projected onto the subspace spanned by independents in y by applying the Fourier-Motzkin algorithm and then omitting the bounds on the other independents, namely those in i. This leads to bounds on the independent components of y (shown in the bottom right of the diagram).

In addition, we must also include inequalities representing the equations that relate the dependents in y to the independents. For example, $y_2 = y_1$ in Figure 5.4 is written as a pair of inequalities in y_1 and y_2. These inequalities involve only y's components because y's dependents can always be expressed in terms of y's independents without involving the independents in i. Again, this follows from the column echelon form. We explain this with reference to y_2 in Figure 5.4. The row in E for y_2 has a nonzero in the first column, which means that y_2 varies with μ_1. The top nonzero in this column must be above y_2's nonzero. (These two nonzeros in the same column cannot be the same element; if they were, y_2 would have been classified as an independent.) Therefore, the independent in $\begin{bmatrix} y \\ i \end{bmatrix}$ that corresponds to μ_1 must lie above y_2 and thus must be another component of y. In fact, it is y_1. The preceding argument would have applied to any other column in which y_2's row contains a nonzero and to any other dependents in y, had there been any.

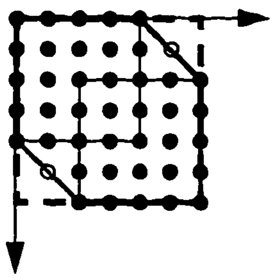
The above two groups of inequalities on y together delineate the image of the loop bounds. Any y such that $y = Fi + f$ for some iteration vector i within the loop bounds is within the image, and vice versa. We outline the argument here. The forward direction of the claim can be justified by "following the arrows in Figure 5.4." Since i satisfies the loop bounds and $y = Fi + f$, the bounds on the independents and hence those derived by projection are satisfied. For the same reason, so are the inequalities relating dependents and independents in y. Conversely, we could start from a vector y in the image and find some i such that i satisfies the loop bounds and $y = Fi + f$. This is done by following the same arrows backward. First, according to Lemma 4.1 on page 87, from y's independents we can pick values for i's independents such that the two together satisfy the bounds on all the

```
FOR i1 = 0, 1
 FOR i2 = 0, 100
  FOR i3 = 0, 100
   ... X2[100*i1+i2,100*i1+i3] ...
```



(a) Ignoring that Iteration Vectors Are Integral

```
FOR i1 = 0, 4
 FOR i2 = 0, 4
  ... X2[i1,i2] ...
  ... X2[i1+2,i2+2] ...
```



(b) Uniformly Generated Accesses

Figure 5.5: Causes of Including Elements Not Accessed

independents. Next, from the independents we can compute i's dependents such that the entire iteration vector i satisfies the loop bounds and $y = Fi + f$.

Imprecision

Our technique, however, has one main limitation. Since we ignore the fact that i, being an iteration vector, must be integral, the set of elements accessed by some iteration may not be calculated precisely. Some elements not accessed may be needlessly included in the restructured array. Figure 5.5(a) illustrates the problem. An element in one of the two empty triangles would be mistaken as being accessed by some iteration because its index vector corresponds to some iteration vector with $0 < i_1 < 1$, which is within the loop bounds 0 and 1 but not integral. Note, however, that because elements are not erroneously omitted, the computed bounds of the restructured array do include all the elements that must be in the array.

5.2.3 Multiple Accesses

Having discussed the case of a single access, we now outline how multiple accesses can be handled. We handle the common case of uniformly generated accesses[4], and fall back to restructuring the whole array in other cases. For uniformly generated accesses, the images of the loop bounds are the same up to a constant offset, as demonstrated by the two squares delineated by thin lines in Figure 5.5(b). Therefore, in each group of corresponding faces from the identically shaped polyhedra, we select the most "lenient" one to form the bounds of the restructured array. In other words, writing the inequalities (one for every access) in the standard form $\beta y \geq \beta_0$, we pick the one with the smallest right-hand side. This results in the larger square bounded by thick dashed lines.

As the figure shows, this may cause unaccessed elements to be mistakenly included in the restructured array. However, we expect that there are many fewer such elements than accessed elements because the offsets are typically small, especially compared with array dimensions. For this reason, only a relatively small amount of memory is wasted.

One may envision more sophisticated analysis, but no major benefits are expected. For example, we could find the image of the loop bounds for each access (whether or not the accesses are uniformly generated) and compute the convex hull of these images. In Figure 5.5(b), this would be the hexagon bounded by thick grey lines. As the figure shows, even this does not guarantee that only accessed elements are included: the elements identified by the two unfilled circles are still included needlessly. Moreover, in the common case of uniformly generated accesses with small offsets, this method saves only a small amount of memory compared with the simpler technique we use.

4. A set of accesses are uniformly generated if their array indices differ only by constant offsets. See previous discussion on page 48.

5.2.4 Implications to Linearization

After computing the image of the loop bounds in the transformed index space, we find a linearization vector with the procedure in Chapter 4, but using as bounds of the restructured array this computed image rather than the transformed array bounds. Though developed with the transformed array bounds in mind, that procedure applies to bounds having the form of a symmetric convex polyhedron, not just transformed array bounds. Therefore, it applies here as well, provided that the computed image is symmetric.

One common case in which the computed image is symmetric occurs when there is only one access and the loop bounds themselves are symmetric, which subsumes the common case of constant loop bounds but not the less common, though still reasonable, case of triangular loop bounds. If the loop bounds are symmetric about, say γ, the restructured array's bounds would be symmetric about $(TA)\gamma + (Ta + t)$. This is because any index vector y within the latter bounds has a (not necessarily unique) corresponding iteration vector i within the loop bounds. The mirror image of i about γ, which is within the loop bounds because of the symmetry, maps to the mirror image of y about $(TA)\gamma + (Ta + t)$, which is therefore within the bounds of the restructured array.

If the image of the loop bounds is not symmetric, we can adapt the linearization procedure slightly to handle a special case, while resorting to restructuring the entire array in others. The adapted procedure differs from the earlier one outlined in Figure 4.4 on page 74 only in the final step, namely computing the relaxed bounds. In particular, the point c is defined exactly as before (i.e., according to (4.24) on page 88), although it is clearly not the center of symmetry — there is no center of symmetry because there is no symmetry. As for the relaxed bounds, in Section 4.5.5 on page 90, they consist of inequalities selected from the transformed bounds and the mirror images of those selected inequalities. Here, all inequalities are from the image of the loop bounds. In particular, for $1 \le j \le m$, we pick the affine functions that yield the lower bounds $l_j(c_1, \ldots, c_{j-1})$, like in Section 4.5.5, and those that yield the upper bounds $u_j(c_1, \ldots, c_{j-1})$, which replace the mirror images before. (Recall that $l_j(\ldots)$ and $u_j(\ldots)$ are augmented bounds for the j-th dimension and are respectively the minimum and maximum of one or more affine functions, as discussed in Section 4.5.3). We check whether each selected pair of affine functions differ only by a constant offset (i.e., they represent parallel hyperplanes). If they do not, the following does not apply, and we fall back to restructuring the entire array. If they do, let the affine function picked from $l_j(\ldots)$ and $u_j(\ldots)$ be respectively

$$\sum_{k=1}^{j-1} F_{jk} y_k + f_j \quad \text{and} \quad \sum_{k=1}^{j-1} F_{jk} y_k + f'_j \tag{5.15}$$

We choose the relaxed bounds as

$$\sum_{k=1}^{j-1} F_{jk} y_k + f_j \le y_j \le \sum_{k=1}^{j-1} F_{jk} y_k + f'_j \quad \text{for } 1 \le j \le m \tag{5.16}$$

or, with F being an $m \times m$ matrix having F_{jk} in the lower triangle and zeros in the main diagonal and upper triangle,

$$f \le (I - F)\, y \le f' \tag{5.17}$$

The relaxed bounds so computed meet the three requirements set out in Section 4.5.5. Let us consider each of them in turn. First, the relaxed bounds have the form required in (4.7) on page 75, with $r_l = f$, $r_u = f'$, and $R = I - F$. This is because, owing to the form of F described above, $I - F$ is a lower-triangular matrix with a unit diagonal. Second, the relaxed bounds include the image of the loop bounds because the inequalities making up the relaxed bounds are all selected from those making up the image of the loop bounds. Therefore, any vector satisfying the latter must, of course, satisfy the former as well. Finally, the relaxed bounds are tight: for any alternative bounds $r'_l \le R'y \le r'_u$ satisfying the above two conditions, $(r'_u - r'_l) \ge (r_u - r_l)$. The argument for this is almost identical to the one given in Section 4.5.5. We can follow the same argument here because up to (4.38) on page 93, it does *not* rely on c being the center of symmetry, only on the definition of c in (4.24), which we also use here. Here, however, the "distance" between the pair of parallel hyperplanes for the j-th dimension is

$$\begin{aligned} u_j(c_1, \ldots, c_{j-1}) - l_j(c_1, \ldots, c_{j-1}) &= \left(\sum_{k=1}^{j-1} F_{jk}c_k + f'_j\right) - \left(\sum_{k=1}^{j-1} F_{jk}c_k + f_j\right) \\ &= (f'_j - f_j) \\ &= (r_{uj} - r_{lj}) \end{aligned} \tag{5.18}$$

Continuing the previous argument from (4.38), we see that $(r'_{uj} - r'_{lj}) \geq (r_{uj} - r_{lj})$, as we want to show.

5.2.5 Summary

A loop may access only those elements whose array indices lie within a limited range. We can use this range, instead of the transformed array bounds, to compute a linearization vector for the restructured array using the procedure in Chapter 4, provided the range is a symmetric convex polyhedron. We adapt the linearization procedure slightly to deal with some special asymmetric cases. To find that range for one access, we map the loop bounds through the transformed access function into the transformed index space. Calculating this image is complicated by the fact that the mapping is not always invertible. We can deal with the problem by casting it as a problem of polyhedral projection. Alternatively, we can "invert" the mapping function in a loose sense: the iteration vector is expressed in terms of part (possibly but not necessarily all) of the array index vector and other independent parameters. The set of accessed elements computed may be imprecise in some cases. However, errors are always on the conservative side: elements that are not accessed may be needlessly put in the restructured array, but elements that are indeed accessed are not incorrectly omitted. Whenever our technique does not apply, we fall back to restructuring entire arrays.

5.3 Summary

Restructuring entire arrays is not always necessary because only elements accessed by the loop need to be in the restructured array. In this case, we may restructure part of an array. We use affine, possibly non-integral index transformations to represent the fact that the restructured array contains only elements regularly spaced in the index space. Runtime index computation still involves only integer arithmetic despite the use of non-integers in the transformations. Moreover, the accessed elements may also be restricted to a region in the array index space. We define the bounds of the restructured array with the image of the loop bounds in the transformed index space but follow basically the same procedure as before to compute a linearization vector.

Part III

Experimentation

Chapter 6

Implementation

We have implemented a prototype compiler based on the SUIF compiler from Stanford University [Wilson et al. 1994]. SUIF is designed for rapid prototyping of compiler techniques. It has been chosen for our implementation because of its extensibility.

The overall structure of our prototype is shown in Figure 6.1. SUIF itself consists of compiler passes communicating through a well-defined intermediate program representation. Each pass reads the program representation from a file, modifies it or annotates it with additional information, and outputs the result. SUIF accepts C or Fortran programs. A frontend converts the source program to its intermediate representation[1]. This representation is then analyzed and transformed by various passes. Finally, code is generated from the transformed representation. SUIF has a MIPS code generator. To compile for other architectures, it can also generate C code, which may then be compiled by the native C compiler of the machine platform on which the binary is eventually executed.

To the basic SUIF infrastructure we have added two components for array restructuring: compiler analysis and runtime support. Compiler analysis is implemented in an array restructuring pass immediately before code generation and after other existing passes. (The rationale for and limitations of this decision are discussed in Section 6.2.2). Runtime support is implemented in a runtime system linked with the generated code. The following sections discuss these two components in detail.

1. Fortran code is translated to C before being read by SUIF proper. A separate SUIF pass recovers certain Fortran-specific information lost in the translation by recognizing idiomatic patterns in the C code that the translator produces.

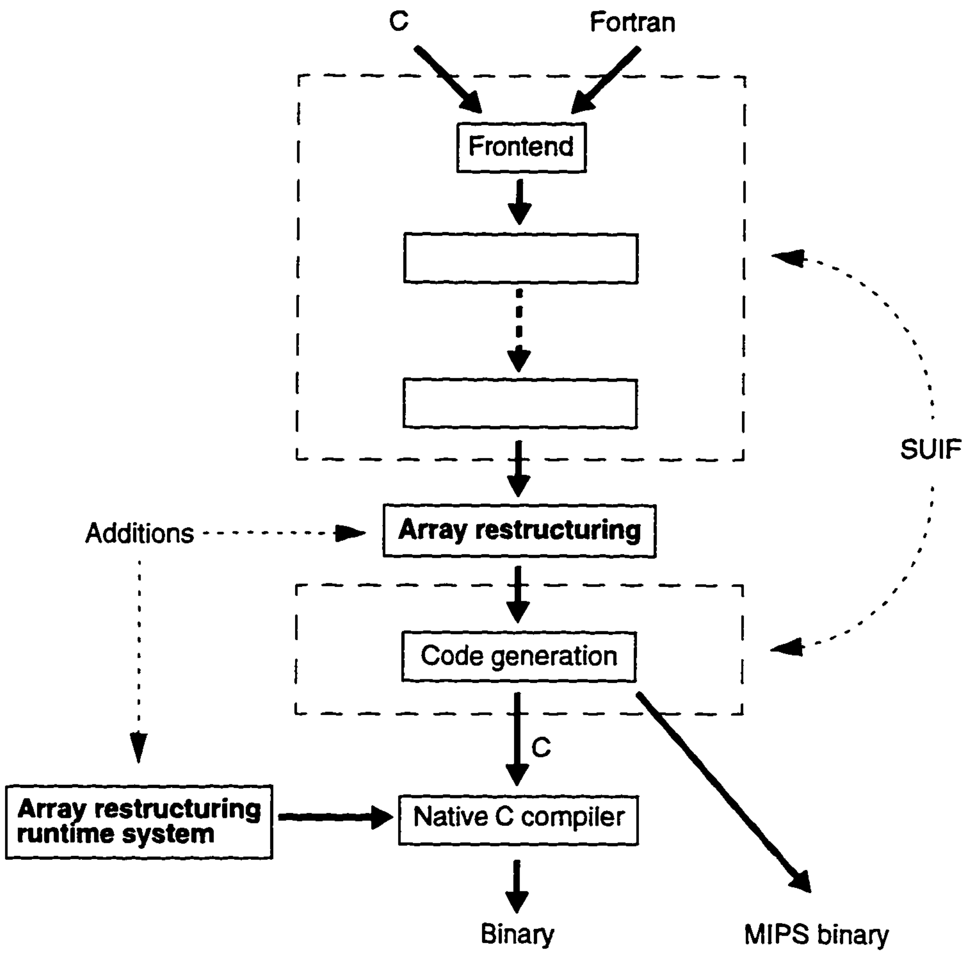


Figure 6.1: Organization of Implementation

6.1 Runtime System

The runtime system is called immediately before the execution of each loop nest. It has two main functions: managing copies of each array and dealing with runtime constants.

The runtime system manages multiple copies of the same array to support a lazy restructuring scheme. (We summarize here previous related discussion on page 63 in Section 3.3.4.) In this scheme, the compiler determines for each loop how an array should be transformed. At run time, multiple copies of an array transformed differently may exist simultaneously. In preparation for loop execution, the runtime system finds a copy with the desired transformation, creating one if there is none. Elements are copied from a valid copy using data remapping routines that are carefully coded for maximum efficiency. The runtime system also ensures consistency between array copies by invalidating all but the one being used if it is written.

Our runtime system also deals with runtime constants, such as loop bounds and array bounds, which typically depend on the problem size and thus the input, as in the example in Figure 6.2(a). The compiler needs the array index expressions to select index transformations. This is possible because the strides in those expressions are usually known at compile time. Given the transformation, the compiler can then modify the array accesses. However, the linearization vector is not fully determined yet because computing the bounds of the restructured array requires runtime information like the original array bounds, the loop bounds, or both. Thus, our compiler calculates as much as it can with what is known and outputs the intermediate results into the generated code. Before loop execution, the runtime system in effect completes the process and calculates the linearization vector, combining information from compiler analysis and runtime information that has become available.

6.2 Array Restructuring Pass

The array restructuring pass implements the analysis techniques presented in previous chapters. In this section, we describe its major functions and discuss its limitations.

6.2.1 Functions

The array restructuring pass relies on SUIF to identify the loops, array declarations, and array accesses and to provide information on the loop bounds, array bounds, and index expressions, among other things. Given this information, it makes restructuring decisions and modifies the program representation accordingly.

Making Decisions

The array restructuring pass makes two types of decisions. First, it analyzes the access pattern of each loop nest and selects an index transformation for each array. This has been discussed extensively in previous chapters.

Moreover, it also decides whether to restructure an array at all according to a heuristic comparison of the cost and benefit. In the current implementation, our simple heuristic is based on the assumption that the runtime cost of restructuring an array, mainly the cost of copying elements, is roughly proportional to the array size, whereas the benefit is roughly proportional to the (dynamic) number of accesses to that array. Thus, array restructuring seems profitable for, say, a triple loop accessing a two-dimensional array because each element is likely to be accessed many times but copied only once (or twice if the array is written).

However, this view may be deceptive. First, while the array is two-dimensional, the loop may access only one dimension, say a row, of it. Second, while an access may appear syntactically within a triple loop, its index expressions may vary with only the outer loop variables. In this case, the benefit of array restructuring does not grow with how many times the innermost loop repeatedly accesses the same element. In fact, a good compiler would lift the array access outside the innermost loop and replace it with a scalar variable inside, a common optimization called scalar replacement [Bacon, Graham, and Sharp 1994]. Therefore, our heuristic rule has to look beyond the number of array dimensions or loop levels.

The prototype compiler considers array restructuring profitable if the access matrix has more columns, excluding zero columns at the right, than its rank. The former is an indication of how often the access is executed; the latter is the dimensionality of the subspace of elements accessed. Neither is affected by left-multiplying the access matrix with a nonsingular matrix, as we do when restructuring an array. In other words, the heuristic yields the same result whether it is applied to the original or transformed access matrix. Although this simple rule has been effective so far, we expect that a more accurate cost-and-benefit analysis using information on actual loop and array bounds to be required for larger, more complicated applications.

Modifying Code

After deciding whether and how to restructure each array, the array restructuring pass modifies the program accordingly. Specifically, it transforms array accesses and inserts calls to the runtime system. Figure 6.2 shows sample output code, abbreviated and slightly modified for presentation, for one of the subroutines used in our experiments.

Accesses to the restructured array replace those to the original, as illustrated by the code shown in bold in Figure 6.2. We treat the restructured array as one-dimensional regardless of the dimensionality of the original. (For example, `a2` and `b2` in Figure 6.2(b) are pointers to elements.) This is because its bounds cannot always be expressed in a conventional multidimensional array declaration, as noted in Section 4.1. Besides, since we compute the linearization vector directly, we may as well treat the restructured array as a one-dimensional array. The lone array index is the scalar offset into the array expressed directly as a dot product of the linearization vector and the loop variables (see code in bold). Note that some components of the linearization vector are variables because they cannot be fully determined until run time, as explained in Section 6.1.

The array restructuring pass also inserts calls to the runtime system (the calls to `Transform` and `Cleanup` in part (b) of the figure). They are placed where arrays may be dynamically restructured: immediately before loop nests and before procedure returns. The latter is for restoring restructured arrays back to their original, canonical layouts before returning. Note that calls to the runtime are needed neither at procedure entry nor after a loop nest (even if some arrays are written). This is because array restructuring is

```
      SUBROUTINE CHOLSKY (IDA, NMAT, M, N, A, NRHS, IDB, B)
      REAL A(0:IDA,-M:0,0:N), B(0:NRHS,0:IDB,0:N)
C
C   CHOLESKY DECOMPOSITION
C
      DO 1 J = 0, N
        ... Update array A ...
1     CONTINUE
C
C   SOLUTION
C
      DO 6 I = 0, NRHS
        ...
        B(I,L,K+JJ) = ...
        ...
6     CONTINUE
      RETURN
      END
```

(a) Original Fortran Code

```
extern int cholsky_(ida, nmat, m, n, a, nrhs, idb, b)
...
{
  float *a2, *b2;                        /* restructured arrays */
  void *descriptor_a; *descriptor_b;     /* array descriptors */
  int lv_a_1, lv_a_2, lv_b_1, lv_b_2;    /* linearization vector */

  a2 = Transform(&descriptor_a, a, ..., lv_a_1, lv_a_2);
  for (j = 0; j <= n; j++) {
    ... /* No need to restructure A. Use original array. */ ...
  }

  a2 = Transform(&descriptor_a, a, ..., lv_a_1, lv_a_2);
  b2 = Transform(&descriptor_b, b, ..., lv_b_1, lv_b_2);
  for (i = 0; i <= nrhs; i++) {
    ... /* Restructure B. Use original A. */ ...
    b2[lv_b_1 * i + lv_b_2 * (k+jj) + l] = ...
  }

  Cleanup(descriptor_a);
  Cleanup(descriptor_b);
}
```

(b) After Array Restructuring Pass

Figure 6.2: Sample Output Code

done lazily. For example, after a restructured array has been written, it is not copied back to the original until execution encounters a loop nest that demands the original array. For the same reason, the runtime system is called even when the compiler has decided not to restructure an array, such as before the first loop nest, because the runtime still needs to update the original array if it does not contain the latest changes. The overhead of making such runtime decisions is trivial compared with the costs of loop execution and, to a lesser extent, data copying.

6.2.2 Limitations

Though useful as a proof of concept in this study, our current prototype leaves several issues for future work. These include alias analysis, interprocedural analysis, and the integration of array restructuring with other compiler optimizations, especially loop restructuring techniques.

First, our prototype does not perform alias analysis. For example, it does not check for aliasing arising from COMMON and EQUIVALENCE constructs in Fortran or pointer-based array accesses in C. Instead, we simply assume that arrays are always accessed through easily identifiable array accesses (never through address pointers, for example) and that distinct array variables refer to distinct, non-overlapping arrays.

However, alias analysis is necessary for array restructuring in general. To restructure an array, the compiler should identify and transform all accesses to that array in the loop nest. This is critical if the array is written. It is less important for read-only arrays because accesses to either the original or restructured version are acceptable, provided that both versions are consistent to begin with. Aliases hinder the task of identifying all accesses to a given array and thereby limit array restructuring.

We omitted alias detection because it is an independent problem beyond the scope of this study. It impacts loop restructuring equally, if not more. Specifically, if aliasing precludes the compiler from restructuring an array, most likely the loop nest itself cannot be restructured either because the same problem would likewise hamper the dependence analysis required for validating loop transformations. In fact, loop restructuring is even more susceptible to such difficulties. For one thing, it is frustrated by aliasing problems in any of the arrays written by the loop, whereas for array restructuring, problems in one array does not preclude the restructuring of another. Also, for array restructuring, we only need to know which accesses are made to the array in question; we need not fully understand how all array indices vary for us to apply a transformation (though such knowledge surely helps in choosing one). To validate a loop transformation, however, we need dependence analysis that demands much more information on the array indices.

Second, the prototype does not perform interprocedural analysis for array restructuring. If a loop nest contains procedure calls, the prototype compiler assumes, perhaps over-optimistically, that the callees do not access any array that is restructured in the caller. Like alias analysis, interprocedural analysis is needed for both loop and array restructuring to deal with procedure calls within loop nests. For array restructuring we only have to determine whether the callee accesses a certain array, whereas for loop restructuring we also need detail information on individual accesses for a full dependence analysis.

Finally, it is to some extent an *ad hoc* decision to perform array restructuring after other compiler optimizations. Exactly how best to integrate array restructuring with other optimizations, especially loop restructuring, warrants much more study than what can be devoted to it in this study. As a first step, we do it after other SUIF optimization passes because the latter obviously were not designed with our array restructuring technique in mind. Therefore, it is better to let the array restructuring pass, which is done last, tackle unintended artifacts of existing passes than vice versa.

Chapter 7

Experimental Results

We performed a series of experiments to evaluate our array restructuring technique. These experiments have been designed to study the performance impact of array restructuring by itself, as well as how it compares and interacts with existing loop restructuring techniques. In this chapter, we report and discuss these results.

7.1 Experiments

We selected for our experiments loops commonly used in related loop restructuring literature and with readily available source code, since we are particularly interested in how array restructuring compares and interacts with existing loop restructuring techniques. These loops include, among others, the NASA7 kernels[1] of the SPEC 92 benchmarks [Dixit 1992], which have been used in several key studies of compiler-directed cache optimizations [Carr, McKinley, and Tseng 1994; Li 1993; Wolf 1992]. They are described in Table 7.1.

For each loop, we experimented with a range of problem sizes. We chose the problem sizes so that the major arrays are at least a few times larger than the second-level cache. This ensures that the data do not fit in the cache. If they did, we would not be able to study

1. Among the SPEC 92 benchmarks, the NASA7 kernels are likely to benefit most from locality optimizations of any kind. In an extensive experimental study, Carr et al. found that, among the SPEC 92 benchmarks, their locality optimization techniques improved performance only for some NASA7 kernels [Carr, McKinley, and Tseng 1994]. Their cache simulation results indicated that before any optimization, only the NASA7 kernels had a significant cache miss rate (slightly below 20%) while other SPEC92 benchmarks' miss rates were no more than 3%, many practically zero.

Table 7.1: Loops for Experiments

Loop	Description[a]	Related Studies
MATMUL	Simple dense matrix multiply. Its innermost loop computes one element of the result matrix.	Cierniak and Li 1995; Kennedy and McKinley 1992; Li and Pingali 1992; Wolf and Lam 1991a
SYR2K	Symmetric rank-2k update for banded matrices. It computes $Z = Z + X^t Y + Y^t X$.	Li 1993; Li 1995; Li and Pingali 1992
MXM	Hand-tuned matrix multiply. Its outermost loop is unrolled four times and jammed.	Carr, McKinley, and Tseng 1994; Li 1993; Wolf 1992
GMTRY	Gaussian elimination. It sets up a linear system for a vortex method solution and inverts the resulting matrix using Gaussian elimination without pivoting.	Carr, McKinley, and Tseng 1994; Li 1993; Wolf 1992
CFFT2D	Two-dimensional fast Fourier transform (FFT). It consists of two routines performing FFT on the first and second array dimension respectively.	Carr, McKinley, and Tseng 1994; Li 1993; Wolf 1992
CHOLSKY	Cholesky decomposition and solution. It performs Cholesky decomposition on multiple banded matrices stored as a three-dimensional array. It then performs forward and backward triangular solves simultaneously for multiple right-hand sides stored as a three-dimensional array.	Carr, McKinley, and Tseng 1994; Li 1993; Wolf 1992
BTRIX	Block tridiagonal matrix solution. It performs a block tridiagonal matrix solution along one dimension of a four-dimensional array.	Carr, McKinley, and Tseng 1994; Li 1993; Wolf 1992
VPENTA	Pentadiagonal inversion. It simultaneously inverts three pentadiagonal matrices.	Anderson, Amarasinghe, and Lam 1995; Carr, McKinley, and Tseng 1994; Li 1993; Wolf 1992

Table 7.1: Loops for Experiments (Continued)

Loop	Description[a]	Related Studies
EMIT	Vortex emission. It is extracted from a vortex code. It creates new vortices according to certain boundary conditions.	Carr, McKinley, and Tseng 1994; Li 1993; Wolf 1992

a. The descriptions for the NASA7 kernels (MXM to EMIT) are mostly adapted from a document distributed with the source code [Bailey and Barton 1986].

how various locality optimizations (either by loop or array restructuring) affect performance because virtually all array accesses would be cache hits. In particular for the NASA7 kernels, our problem sizes differ from, and typically exceed, the standard ones. (The enlarged data sets were simply generated at random because in all these loops the access pattern does not depend on the contents of arrays, only on their sizes.) Since our problem sizes differ from those of the standard benchmarks, our results should *not* be treated as a report on SPEC benchmark performance.

These loops are relatively small pieces of code, though many of them (in particular the NASA7 kernels) have been extracted from complete numerical scientific codes. As explained above, we chose them to facilitate a comparison between array and loop restructuring. Moreover, compared with large applications, they offer two advantages. First, their manageable sizes allow deeper understanding of the observed performance behavior: it is feasible to examine each loop and array to determine what transformations have been used and how they affect the access pattern. Second, our experiments comparing array and loop restructuring required manually applying loop transformations, for reasons to be explained later. This would have been impractical if the applications had been too large.

Naturally, measuring code fragments rather than complete applications has its drawbacks. For one thing, without considering large, complete applications — and as many of them as possible — we cannot tell *how often* array restructuring (or any program restructuring technique for that matter) benefits loop execution, but only *how much* it does for specific loops. Moreover, we cannot accurately gauge the interaction between a loop that array restructuring has transformed and other parts of the program, especially those executed immediately before and after it.

In the latter respect, however, we address the limit of our methodology by making pessimistic assumptions, so that if the measurements are in any way deficient, they are biased against array restructuring. We report performance as the execution times of the loops themselves plus all copying overhead (from some canonical array layout to the desired layout, and back in the case of a read-write array). In effect, this assumes that in the context of a complete application, the overhead is paid whenever execution enters or leaves the loop in question. If it is not the case (e.g., the cost is amortized over multiple loop executions), performance would exceed what our results suggest. Also, in a complete application, a loop's performance is affected by what data happen to be left in the cache by preceding accesses, while what the loop itself leaves in the cache may likewise affect the performance of subsequent execution. We expect this effect to be inconsequential in the experiments because most arrays are at least several times of the cache size and accessed many times. Therefore, the cost of filling even the entire cache at such transitions, both on entry into and exit from the loop nest in question, is small relative to the total cost of accesses inside the loop nest.

All experiments reported in this chapter were done on a DEC 3000 Model 400 workstation based on the Alpha 21064 processor [DEC 1992; Dutton et al. 1991; Sites 1992]. (Results on an IBM RS/6000 Model 41T workstation are included in Appendix B. We omit them here since they are qualitatively the same as those reported below.) Our configuration has two levels of cache for data: an on-chip, 8 KB, write-through data cache, and an off-chip, 512 KB, write-back, unified cache shared by data and instructions. Both levels of cache are direct-mapped. Cache lines are 32 bytes long. On a read miss in the first-level cache, 5 cycles are required to read the accessed data from the second-level cache and another 5 cycles to fill the rest of the cache line. A write miss adds another 5 cycles. On a miss in the second-level cache, it takes 24 cycles to read the accessed data from main memory and another 6 cycles to fill the other half of the cache line. The Alpha 21064 processor also has a 32-entry, fully associative translation lookaside buffer (TLB) for data pages.

The loops were compiled by our prototype compiler. The output C code was compiled by the native C compiler (`cc`)and linked with our runtime system. Standard compiler optimizations (`-O2`) were enabled in both steps. In addition, we augmented SUIF with two simple passes implementing limited versions of two known optimizations unrelated to array restructuring. Both were applied in all cases, independently of whether any loop or array was restructured. The first optimization is scalar replacement [Bacon, Graham, and Sharp 1994]. This optimization replaces an array access in the innermost loop(s) with the use of a temporary scalar variable, whose value is loaded from the accessed element before the loop and stored back afterward if modified. The second optimization replaces loop-invariant upper loop bound expressions in the generated C code with temporary variables storing the pre-computed values to avoid evaluating the expressions in every loop iteration, as implied by the semantics of the `for` construct in C. We added these two optimizations because they proved critical to performance (with or without loop or array restructuring) in some of the loops but, for some reason, were not performed by SUIF or the native C compiler on those loops. Omitting them would have unduly and significantly impacted performance in a way unrelated to either loop or array restructuring, making it much more difficult to assess their effects.

7.2 Array Restructuring

The first, most natural question is whether array restructuring improves performance at all, especially in view of the potential for substantial copying overhead at run time. Our experiments showed that it did in many of the cases. In this section, we also discuss what array transformations the compiler applied in individual cases and give insights into how they improved performance in sometimes unanticipated ways.

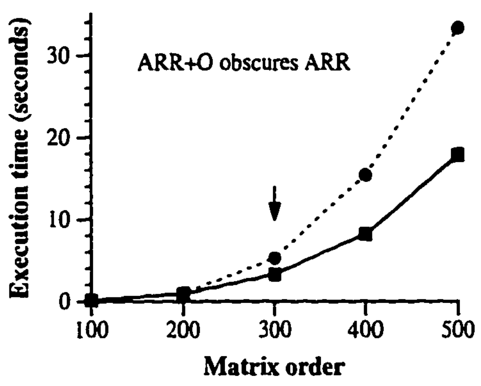
7.2.1 General Results

The loops listed in Table 7.1 fall into three categories according to our compiler's array restructuring decisions. Recall that the compiler heuristically decides whether or not an array should be restructured by comparing the potential benefit and cost. For five of the loops, it decided that array restructuring could be applied profitably despite the potential overhead. Their performance is shown in Figure 7.1. For another three, the compiler determined that loop execution would benefit from array restructuring, but the copying overhead might be too large to justify it. Performance of these loops are shown in Figure 7.2. Finally, for one of the loops (specifically EMIT of the NASA7 kernels), the compiler saw no opportunity to improve the original array layouts. Therefore, no array was restructured. Previous studies likewise found virtually no improvement using other program restructuring techniques [Carr, McKinley, and Tseng 1994; Li 1993; Wolf 1992]. For this reason, we did not measure the performance of this loop; it is omitted throughout the following discussion.

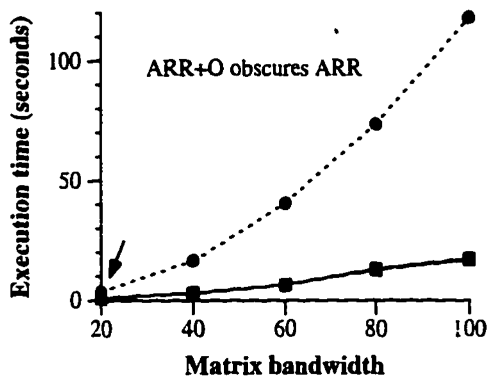
Both Figure 7.1 and Figure 7.2 show how execution times vary with problem sizes for the original loop (i.e., with standard compiler optimizations but no loop or array restructuring) and for array restructuring — one curve for loop execution alone and one including runtime overhead. This overhead includes the cost of copying elements from the original to the restructured arrays, and back if necessary. Time spent by the runtime system to complete the analysis (see Section 6.1) is also attributed to the runtime overhead. In practice, however, it is trivial compared with the copying cost.

First, let us consider Figure 7.1. Generally, array restructuring improved performance substantially. In all cases except SYR2K, execution times were roughly halved. For SYR2K, the improvement was more dramatic: execution times decreased by almost a factor of seven.

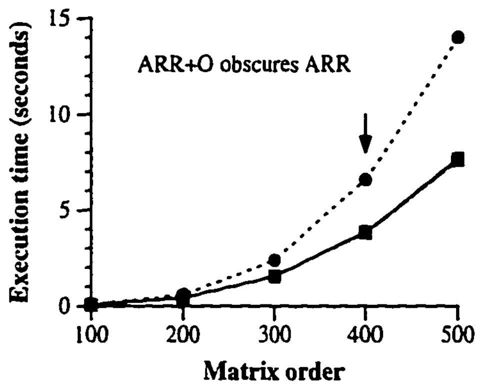
GMTRY, CFFT2D, and CHOLSKY contain some loop nests for which our prototype compiler decided that the default array layouts were already best and hence did not



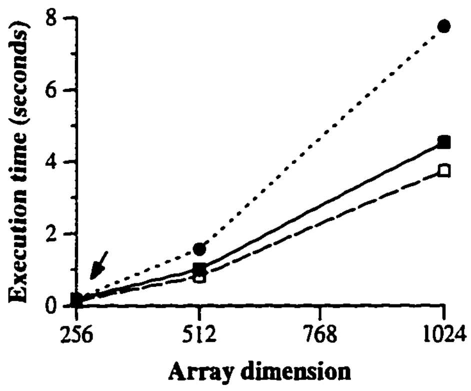
(a) MATMUL



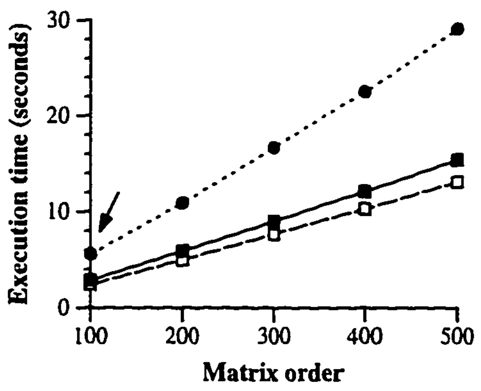
(b) SYR2K



(c) GMTRY



(d) CFFT2D



(e) CHOLSKY

··●·· Original loop (BASE)
—□— Array restructuring (without overhead) (ARR)
—■— Array restructuring (with overhead) (ARR+O)
——► points at size threshold for L2 cache

As noted in graphs, some curves may obscure others.

Figure 7.1: Array Restructuring Performance — Profitable Cases

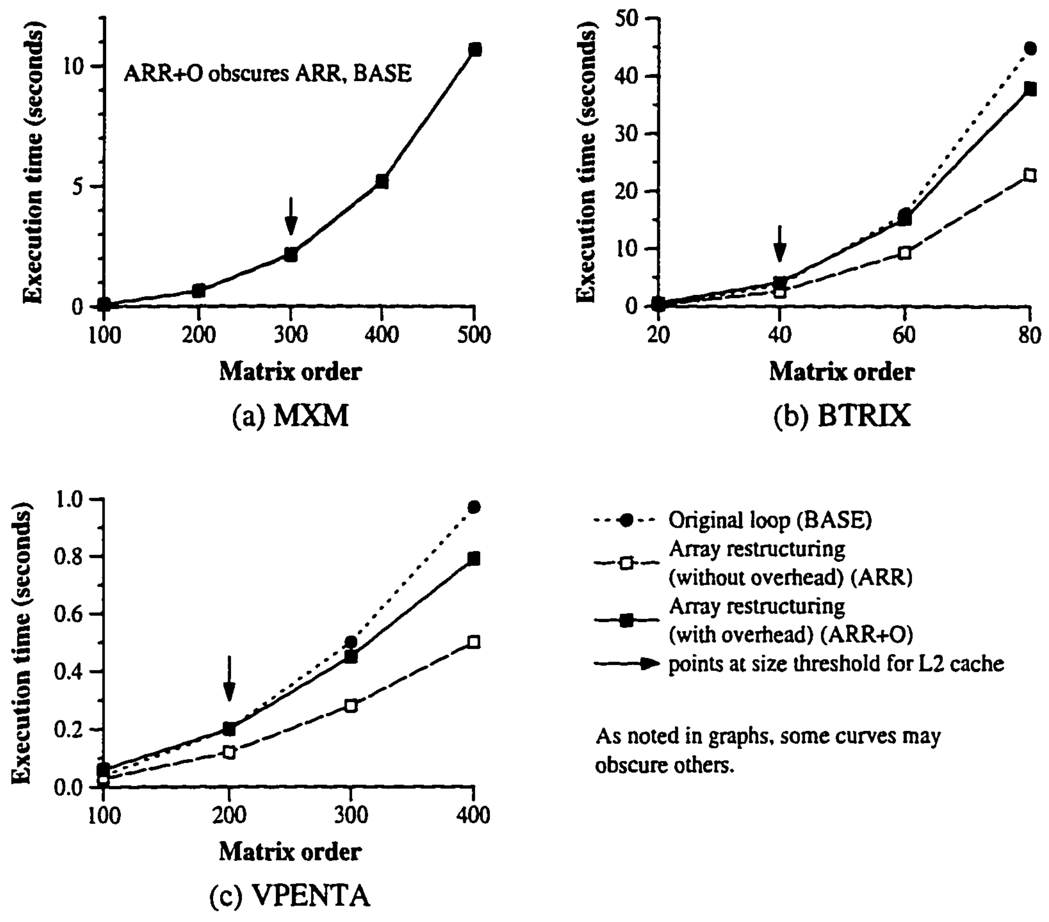


Figure 7.2: Array Restructuring Performance — Unprofitable Cases (with Manual Override of Compiler Decision)

restructure any array. The execution times shown here and in the remainder of this dissertation do not include contributions from such loop nests (though we did compile and execute the entire routines in all experiments). This allows us to see clearly the direct performance impact of array restructuring when it was indeed applied. The results shown here are qualitatively representative of the total execution times, however, as the part that array restructuring did affect dominated the whole computation in each case. For example, Figure 7.3 shows the execution times of the complete routines, including both the parts affected and unaffected by array restructuring. These timings exhibit the same trend as

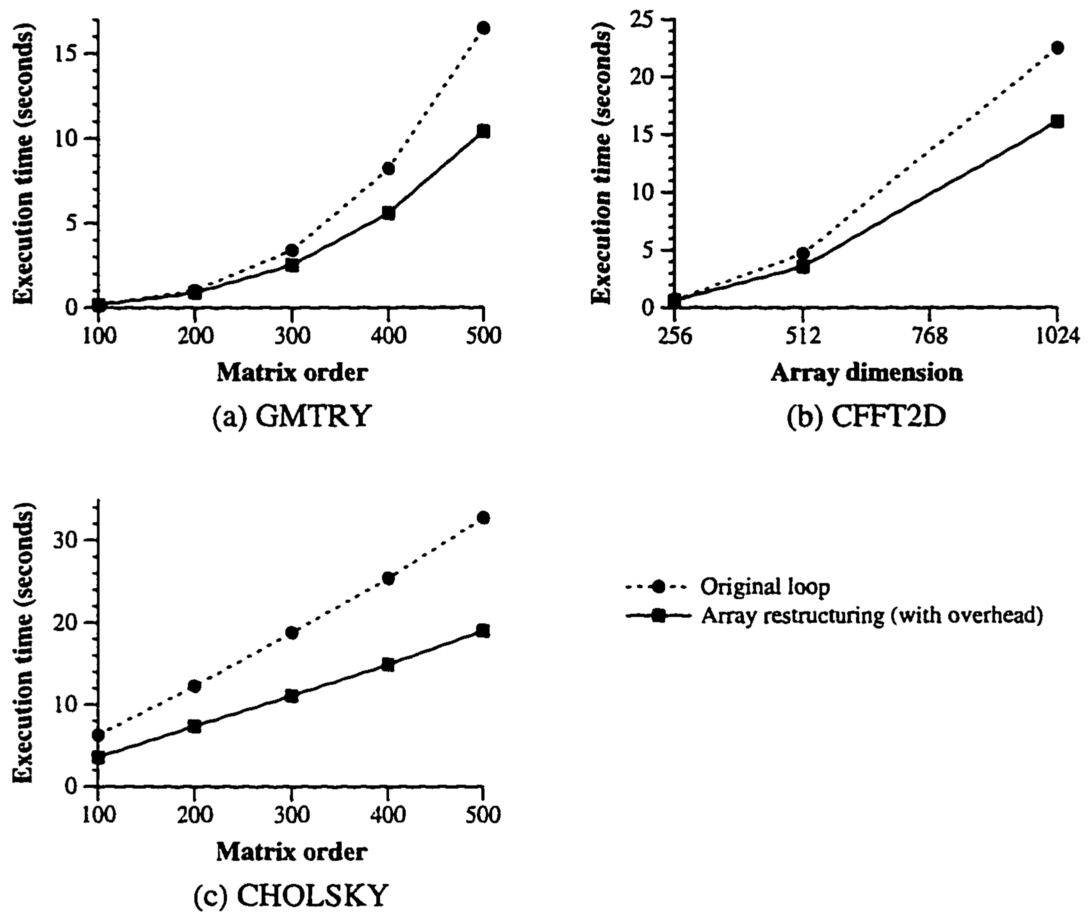


Figure 7.3: Array Restructuring Performance — Complete Execution Times

those for the affected part alone (in Figure 7.1), although the percentage improvement is, of course, lower because of contributions from the unaffected loop nests.

Performance improved despite the runtime overhead. For MATMUL, SYR2K, and GMTRY, the overhead was negligible. In fact, it was so small that including the overhead did not visibly change the performance curves. In the cases of CFFT2D and CHOLSKY, the overhead did have a noticeable impact. However, it remained much smaller than the performance gain in loop execution. Therefore, array restructuring still increased overall performance significantly even though the overhead was nontrivial.

Furthermore, performance improvement was consistent over a wide range of problem sizes, not just for particular sizes. We expect this to hold true in problems larger than what we measured: array restructuring would still improve performance substantially; the overhead would still have little impact. In fact, the overhead would decrease relative to execution times. This is because asymptotically the overhead grows more slowly than the performance gain. The copying overhead is roughly proportional to the amount of data copied, and thus to the size of the arrays being restructured (assuming, in the worst case, that entire arrays are restructured). The performance gain, however, is expected to grow at least linearly with the computation (or, more precisely, the part involving the array accesses in question). In these five loops, the total array size grows asymptotically slower than the computation. Our cost-and-benefit heuristics attempt to designate only arrays and loops with this property as candidate for array restructuring.

Figure 7.2 shows the execution times of loops for which our compiler decided not to restructure any array because of concerns over copying overheads. Manually overriding the compiler to restructure arrays once for the entire routine, we obtained the measurements shown in the figure to see whether that decision is correct and how much performance would have suffered if our heuristics had designated these cases as profitable.

Not surprisingly, array restructuring did not improve overall performance in any significant way. For BTRIX and VPENTA, it did roughly halve the execution times of the loops themselves. However, the copying overhead was also substantial, as expected, offsetting most of the gain. Incidentally in these cases, performance was improved by array restructuring despite such overhead, at least for the larger of the problems that we measured (but not some small problems like VPENTA at 100). We expect these observations to hold for even larger problems that we did not measure, because in these loops both the total size of arrays being restructured and the computation involving the accesses in question grow at the same rate.

Finally, Table 7.2 shows the memory overheads of array restructuring, namely the sizes of the restructured arrays. Only a single problem size is shown for each loop as the relative overheads are independent of absolute array sizes. The data exclude arrays that are not restructured. In all cases except SYR2K, the restructured arrays are as large as the original because the compiler decided, in effect, to permute array dimensions. For SYR2K, the index transformation skews the array index space. Some memory is left unused to facilitate index computation, as discussed in Chapter 4. Partial restructuring could have reduced the size of the restructured array to the exact size of the original. However, our current implementation failed to do so because of runtime constants in loop bounds.

Table 7.2: Sizes of Original and Restructured Arrays

Loop	Size of Original Array(s) (MB)	Size of Restructured Array(s) (MB)
MATMUL	0.95	0.95 (100%)
SYR2K	6.10	6.41 (105%)
MXM	0.95	0.95 (100%)
GMTRY	0.95	0.95 (100%)
CFFT2D	8.00	8.00 (100%)
CHOLSKY	9.54	9.54 (100%)
BTRIX	11.6	11.6 (100%)
VPENTA	6.10	6.10 (100%)

To summarize, when our compiler decided that array restructuring would be profitable, performance improved substantially for a wide range of problem sizes despite modest, sometimes insignificant, copying overheads. The compiler also successfully identified cases in which array restructuring would not be profitable because of the overhead. If we had performed array restructuring in those cases, performance would have improved only slightly, if at all.

7.2.2 Individual Cases

Array transformations used in individual cases and the problems they address are summarized in Table 7.3. We discuss in detail only the case of SYR2K because it demonstrates an unanticipated effect of array restructuring on something beyond cache misses.

We believe that array restructuring dramatically improved SYR2K's performance mainly because it reduced TLB misses. Like the reduction in cache misses, this resulted from accessing elements consecutively in memory.

To see this, let us examine the access pattern of SYR2K, shown in Figure 7.4. We consider only the accesses to array `X`; those to `Y` are identical. `X` has long rows and relatively short columns. Both accesses to `X` go through it diagonal by diagonal. Although this suggests, correctly, a need to improve spatial locality, the impact on cache misses is smaller than it seems because cache lines are generally reused soon. Consider the access `X[j-k+b,k]`. Little data reuse is expected between iterations of the innermost loop because they read elements in a diagonal, which are far apart. However, the middle loop carries spatial reuse. When the innermost loop is next executed (as the next middle loop iteration), the access moves on to the next diagonal. The new diagonal's elements are probably in the same cache lines as the old because the two are adjacent. As the innermost loop has relatively few iterations (only tens in the experiments), cache lines containing the old diagonal are likely to remain in the cache. Furthermore, temporal reuse is carried by the outermost loop. In any one iteration of the outermost loop, the access never moves beyond a fixed column, touching a triangle of elements. When we execute the next iteration, this column and hence the triangle shift right by one. Therefore, consecutive triangles overlap considerably: in any one triangle, only elements in the boundary column are not read before. The rest probably have stayed in cache since the last iteration because each iteration has to displace only a small number of cache lines to make room for newly read elements, namely the boundary column. In fact, even an entire triangle contains at most a thousand or so elements, which occupy just a fraction of common cache capacities. As for the access pattern for the other access, `X[i+k-b,k]`, it is similar except that temporal

Table 7.3: Problems and Solutions in Individual Cases

Loop	Problematic Access Pattern[a]	Solution by Array Restructuring
MATMUL	The innermost loop reads one array column by column.	Transpose array so that the innermost loop accesses it row by row.
SYR2K	The innermost loop reads two arrays diagonally.	Skew array index space to map diagonals in the original arrays to rows in the restructured arrays.
MXM	An access in middle loop (of a triply nested loop) reads an array column by column.	Transpose array so that it is read row by row.
GMTRY	Implementation of Gaussian elimination updates a submatrix column by column.	Transpose array so that the corresponding submatrix in the restructured array is updated row by row.
CFFT2D	One of two phases performs FFT on columns; the innermost loop therefore updates a two-dimensional array column by column.	Transpose array dynamically between phases so that it is always updated row by row.
CHOLSKY	The solution phase updates a three-dimensional array for the right-hand sides not in row-major order: the innermost loop increments the middle array index.	Permute array indices so that the array is accessed in "almost" row-major order: the innermost loop increments the last index, middle loops increment middle indices, etc.
BTRIX	Four four-dimensional arrays are not accessed in row-major order.	Permute array indices so that all four arrays are accessed in "almost" row-major order. For one of them, the restructured array contains only the plane of the original that is really accessed.
VPENTA	All six two-dimensional and one three-dimensional arrays are not accessed in row-major order.	Permute array indices so that all arrays are accessed in strictly row-major order.
EMIT	None is found.	None is required.

a. In line with the convention in this dissertation, we describe the problematic access patterns and their solutions in terms of row-major arrays, although the arrays are column-major as the source code is in Fortran. Thus, when we, for example, say that a loop accesses a two-dimensional array (which is implicitly row-major) "column by column," we mean that it strides through the least rapidly varying array dimension. For a column-major array in Fortran, this would mean going through elements of a row. This is only a matter of terminology; the Fortran code itself was *not* modified or affected in any way.

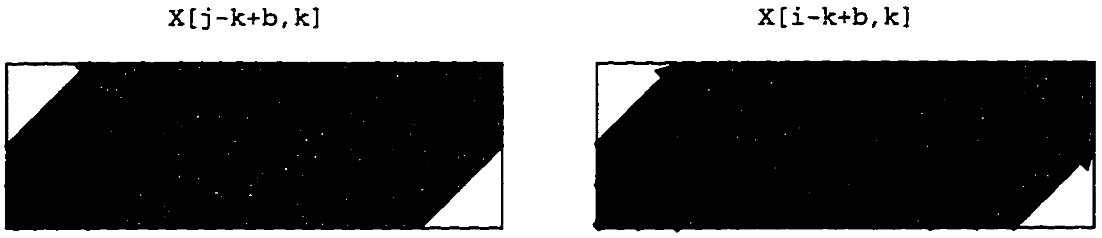
reuse is carried by the middle loop: each outermost loop iteration reads one diagonal repeatedly (hence the thickness of the arrows in the diagram).

Since cache misses do not seem to have enough impact to account for the performance difference observed, we believe TLB misses to be a more likely cause of SYR2K's poor performance. Notice that consecutive innermost loop iterations may touch different pages because they read elements separated by a row long enough to span a page or more. A row, and therefore page, is touched again only after a complete execution of the innermost loop. If there are not enough TLB entries for all the rows in this and other arrays, each access potentially suffers a costly TLB miss. This seems to be our case: the Alpha 21064 processor has 32 TLB entries for data [DEC 1992], fewer than the total number of rows in all the arrays. (Page faults cannot explain the poor performance because our measurements indicated no unusual paging activity.)

Although we had no way to directly measure TLB misses, the following experiment lent support to our hypothesis. We measured the original loop for different values of `b` (see the top of Figure 7.4 for `b`). The results are shown in Figure 7.5. By reducing the value of `b`, we reduce the number of rows the innermost loop touches while keeping each row long enough to need a separate TLB entry. Performance would suffer if there are not enough entries for all the rows touched. For a TLB with 32 entries, we expect the threshold to be 7 or 8 because each execution of the innermost loop touches some elements of `4*b-1` rows (all `2*b-1` rows of array `X`, the same for `Y`, and one row of `Z`). Figure 7.5 shows a discrete change at the expected point: between 7 and 8, execution time rises by 120%, much more than can be explained by the corresponding 30% increase in the number of iterations, and the execution time rises much faster thereafter.

Array restructuring reduced TLB misses by making array accesses go through elements consecutively in memory. The resulting access pattern is shown at the bottom of Figure 7.4. The innermost loop reads a row of elements, which are in adjacent cache lines and likely to be in the same page. Moreover, the triangle of elements touched by one outermost loop iteration covers half the elements in consecutive, short rows rather than a few

```
Declare X[2*b-1,n], Y[2*b-1,n], Z[2*b-1,n]
FOR i = 1, n
 FOR j = i, max(i+2*b-2,n)
  FOR k = max(1,i-b+1,j-b+1), min(n,i+b-1,j+b-1)
   Z[j-i+1,i] += X[j-k+b,k] * Y[i-k+b,k] +
                 X[i-k+b,k] * Y[j-k+b,k]
```



becomes

```
Declare X2[n,2*b-1], Y2[n,2*b-1],Z[2*b-1,n]
FOR i = 1, n
 FOR j = i, max(i+2*b-2,n)
  FOR k = max(1,i-b+1,j-b+1), min(n,i+b-1,j+b-1)
   Z[j-i+1,i] += X2[j+b,-j+k-b] * Y2[i+b,-i+k-b] +
                 X2[i+b,-i+k-b] * Y2[j+b,-j+k-b]
```

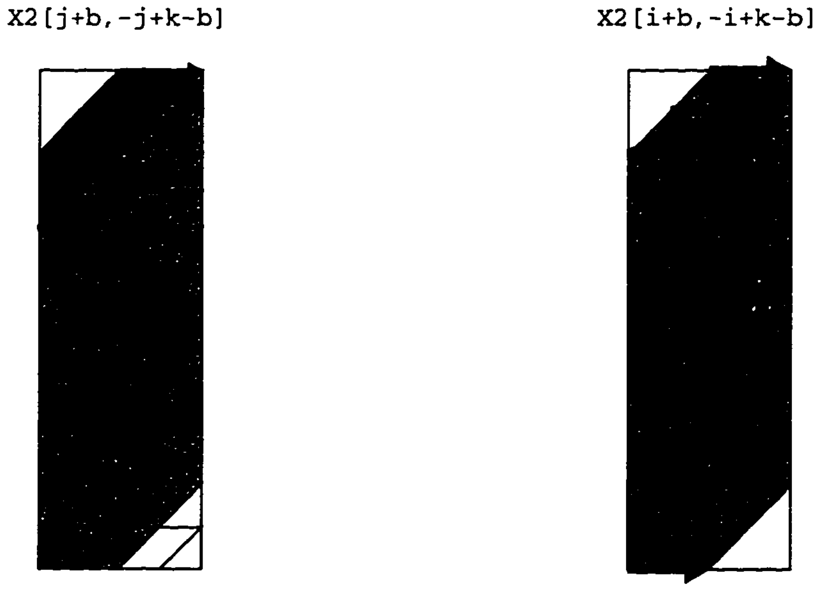


Figure 7.4: Access Pattern of SYR2K

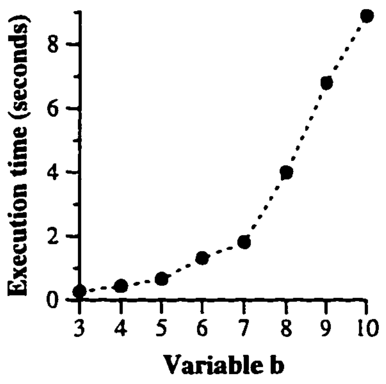


Figure 7.5: Performance Variation of SYR2K

elements per short row (and presumably per page) scattered around the array. As the triangle shifts down the restructured array, only a few TLB entries are needed for the working set at a given time. This would have helped paging behavior as well had it been a problem.

7.3 Comparing Array Restructuring and Loop Restructuring

In addition to evaluating array restructuring itself, we also compared array restructuring with common loop restructuring techniques that also aim at improving locality, spatial as well as temporal. Before discussing the results, we first describe the comparison methodology.

7.3.1 Methodology

We compared the execution time of each loop after array restructuring with that after loop restructuring. To apply array restructuring, we simply compiled each loop with the prototype, manually overriding the compiler's decision not to restructure because of potentially high overhead in some of the loops. On the other hand, loop restructuring was done manually as described below.

Each loop was restructured by manually changing its Fortran source. We then compiled it with SUIF (without array restructuring). In each case, the best loop transformation was chosen manually using various loop restructuring techniques in the literature and their combinations. Transformations were both chosen and applied manually because we would like to consider a wide range of techniques in the literature, but no single compiler to our knowledge has implemented all of them. Moreover, any particular implementation of a general technique might fail to transform a given loop because of limitations incidental to the implementation but not inherent to the technique itself. The transformations considered include loop interchange (or permutation, for loops more than doubly nested) [Allen and Kennedy 1984; Kennedy and McKinley 1992], skewing [Wolfe 1989b], reversal [Bacon, Graham, and Sharp 1994; Wedel 1975], scaling [Li and Pingali 1993c], fusion, and distribution [Carr, McKinley, and Tseng 1994; Kennedy and McKinley 1990, 1994].

To calibrate the quality of the loop restructuring we did by hand, we also compared our manually selected transformations with those performed by the POWER Fortran Accelerator (PFA) on a Silicon Graphics Power Challenge [SGI 1995b]. We found the manual restructuring to be as good as, and sometimes better than, what PFA did. PFA is a source-to-source code optimizer. It performs a number of advanced loop transformations automatically to maximize locality and parallelism. Those relevant to locality include loop interchange, fusion, blocking, and unroll-and-jam [SGI 1995b]. In five of our loops, PFA performed basically the same transformations as we did. In the other three, PFA did not perform any transformation. (Without PFA's source code, it cannot be determined whether PFA did not transform these loops because it could not or because it decided against any change.) In two of these three loops (SYR2K and CHOLSKY), our manual restructuring based on other existing techniques improved performance significantly. In the last remaining case (CFFT2D), we did not transform the loops manually because it seems extremely unlikely that they could be automatically transformed by a compiler with existing technology. The major difficulty is the use of indirection arrays for indexing read-write data arrays, which frustrates dependence analysis.

Experimental results are shown in Figure 7.6. The bar chart shows the execution times of each loop after different kinds of processing, all normalized to the execution time of the original loop. Normalized, rather than absolute, execution times are shown because we wish to consolidate the results for all loops in one chart to facilitate discussion, but the execution times of different loops differ vastly. In the graph, all timings for array restructuring include two components: loop execution and data copying. "Manual loop restructuring" refers to the manually chosen loop transformations; "automatic loop restructuring" refers to those selected by PFA automatically. The latter gives us some indication of what one particular production-quality optimizing compiler based on the latest technology can achieve. Performance resulting from other compilers may, of course, vary. The problem size of each loop is at the top of the range used in previous experiments (which have been reported in Figure 7.1 and Figure 7.2).

7.3.2 Results

Generally, the results suggest that array and loop restructuring complement each other. More specifically, the loops fall into three categories based on the comparison between the two types of techniques.

For MATMUL, SYR2K, and CFFT2D, array restructuring excelled in applicability as well as performance. It either applied where loop restructuring did not (CFFT2D) or produced better performance despite the copying overhead. MXM is a unique case. Array restructuring did not improve performance; it performed better than loop restructuring only because the latter degraded performance "unexpectedly." MXM is a hand-tuned implementation of matrix multiplication in which the outermost loop has been unrolled four times. We deliberately permuted the loops in a way that would be optimal if the unrolling were absent, expecting that a compiler would do the same. The SGI PFA confirmed our expectation.

For CHOLSKY and GMTRY, array restructuring and loop restructuring resulted in comparable performance improvement. For CHOLSKY, however, array restructuring

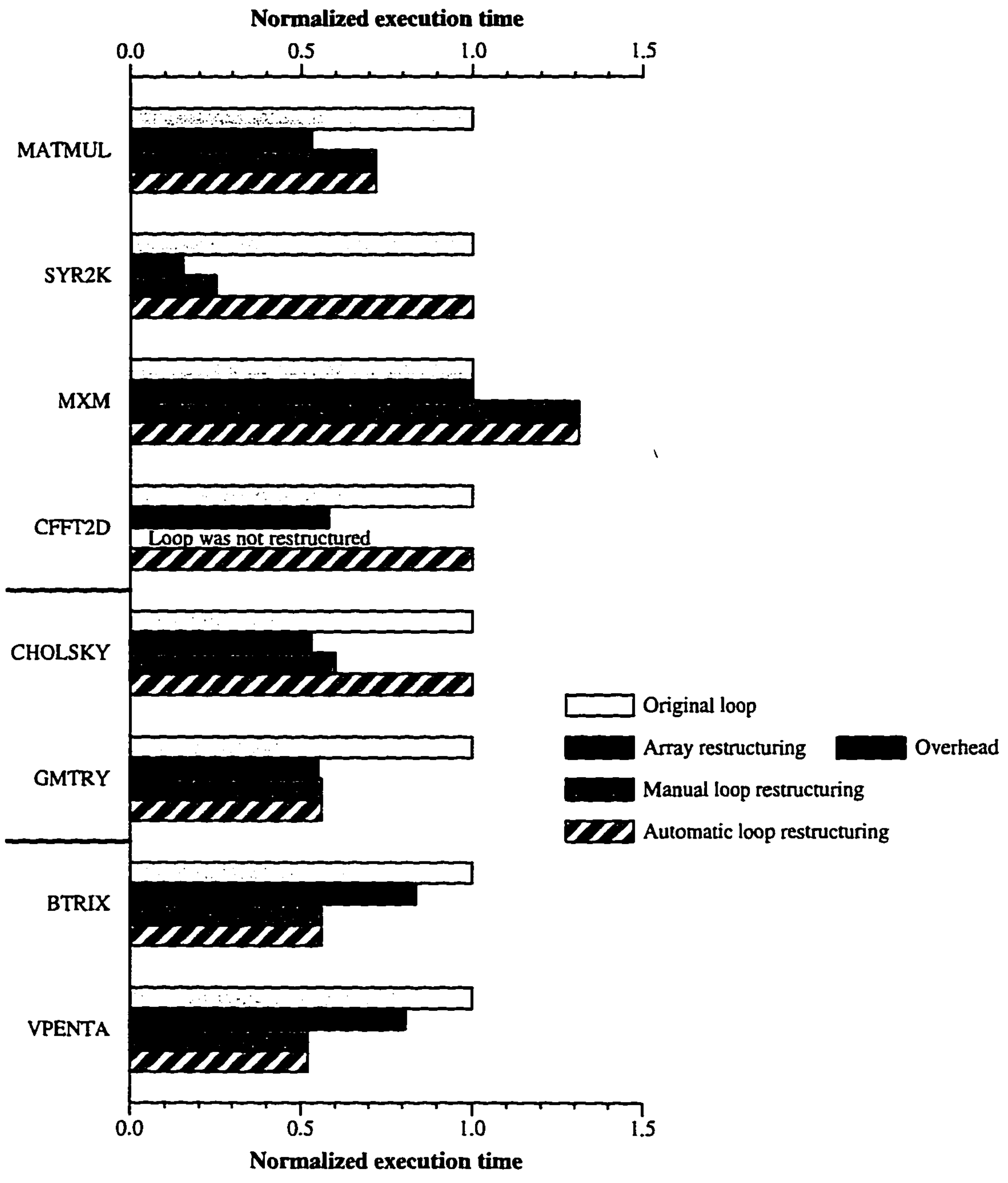


Figure 7.6: Comparing Array and Loop Restructuring

arguably performed better because it seems difficult to automatically select the complicated sequence of loop transformations required to get the reported performance. Difficulties include imperfect nesting at almost every loop level and certain array indices and loop bounds that involve multiple loop variables. The entire sequence was chosen with considerable experimentation and human intuition, even though it merely consists of well-known transformations including loop interchange, fusion, and distribution. It effectively turned the loop nest completely "inside out." The SGI PFA did not transform this loop nest. Therefore, array restructuring achieved better performance than automatic loop restructuring, although it was only comparable to manual loop restructuring.

For BTRIX and VPENTA, loop restructuring was undoubtedly the better approach. Although array and loop restructuring both reduced loop execution times to the same extent, loop restructuring need not incur the runtime overhead of copying array elements. Moreover, the loop restructuring done in both cases was simple: we merely interchanged loops in a few perfect loop nests. The automatic optimizer performed the same with little difficulty.

Finally, we note that the prototype compiler successfully identified the cases where it would be wiser to do something else than to restructure arrays. As mentioned earlier, the compiler decided against array restructuring for three loops: MXM, BTRIX, and VPENTA. From Figure 7.6, we see that for MXM it is best not to restructure anything, loop or arrays. The original version is already highly optimized, at least for the particular architecture in these experiments. (The graph may suggest that array restructuring performed equally well, but in fact it performed slightly worse because of the overhead, although the difference is too small to be visible.) As for BTRIX and VPENTA, some simple loop restructuring suffices to decrease execution times significantly.

To sum up, in these results array restructuring complemented loop restructuring. It applied where loop restructuring did not. When both applied, array restructuring performed comparably, sometimes better. In cases where it did not perform as well, some simple loop restructuring would have been good enough. This strongly suggests that combining the two can potentially improve performance even further.

However, their complex interaction makes exploiting this potential nontrivial. For example, suppose we first apply automatic loop restructuring and then array restructuring. One might expect some extra performance improvement over applying either alone. This, however, will not materialize. As we have noted, out of the eight loops in the experiments, five were automatically transformed by PFA and three were not. In the latter group of loops, naturally, combined loop and array restructuring would have the same performance as array restructuring alone, since it *is* array restructuring alone. As for the former group, our prototype would not transform the arrays any further because they already have good spatial locality after loop restructuring. Worse, for MATMUL and MXM, better performance could have resulted from simply applying array restructuring to the original loop. The same would also happen in the case of SYR2K if the results for manual loop restructuring are considered. The reason for this is that in trying to improve spatial locality, which array restructuring could have done without jeopardizing temporal locality, loop restructuring sacrifices some temporal locality — a loss that subsequent array restructuring cannot recover. More work is required to fully understand how the two types of restructuring should be integrated.

7.4 Array Restructuring and Tiling

Array restructuring also complements loop tiling, a powerful locality optimization technique for nested loops [Lam, Rothberg, and Wolf 1991; Wolf 1992; Wolf and Lam 1991a]. This section discusses how the two interact.

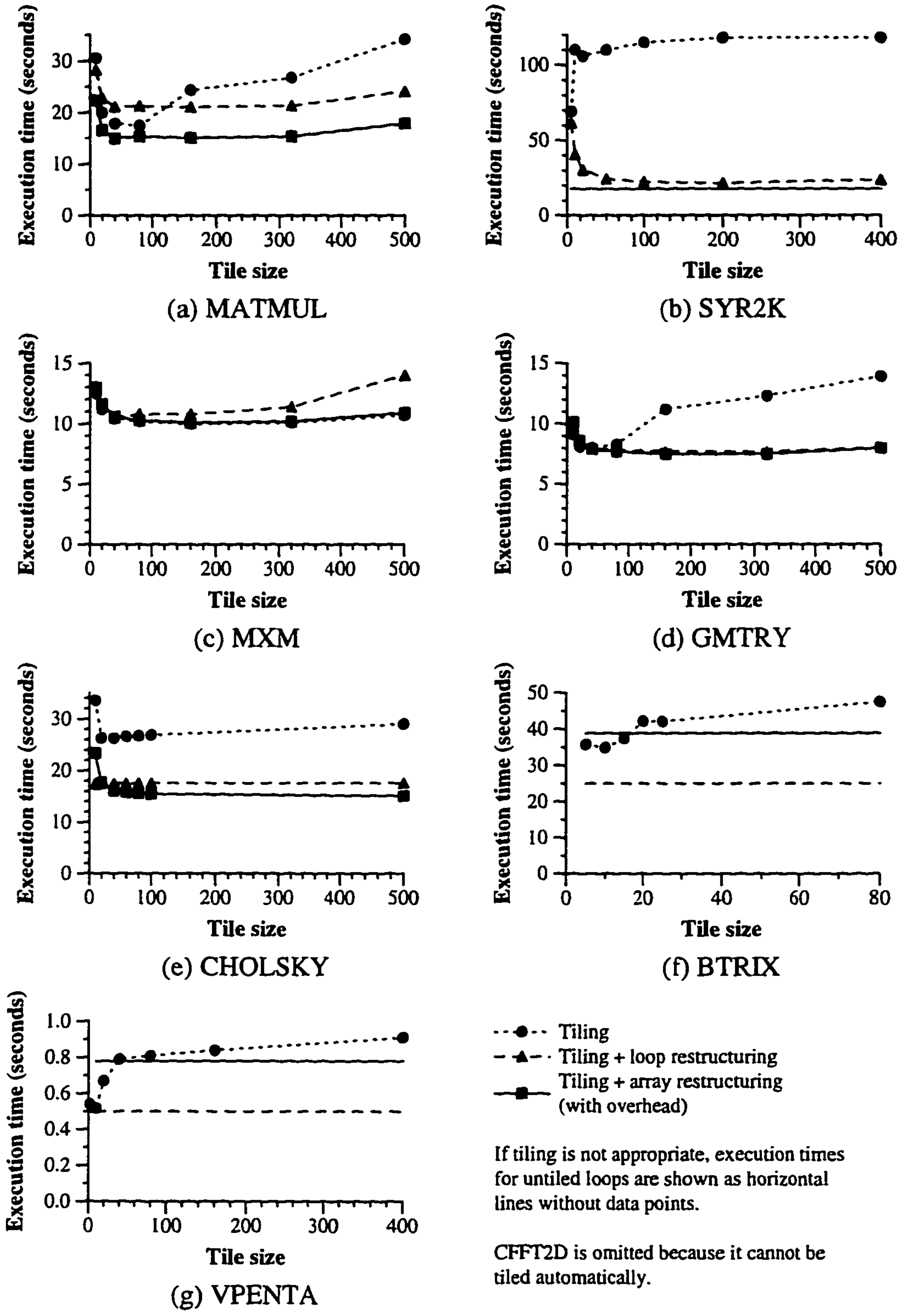
We experimentally compared three forms of each loop: tiling alone, tiling with loop restructuring, and tiling with array restructuring. In each case, the innermost loop(s) were tiled manually by changing the Fortran source[2]. For loop restructuring, the loops were restructured and then tiled, as in previous work [Li 1995; Wolf and Lam 1991a]. For array restructuring, however, we tiled the original loop and then applied array restructuring because we could not manually tile the output code of the prototype compiler. Tiling the loops first also agrees with this dissertation's convention of applying array restructuring last (see Section 6.2.2). We expect results to be similar if array restructuring had been applied first because our algorithm chooses array transformations based on the directions in which inermost loops go through the index space, and loop tiling does not alter those directions. We measured execution times for a range of tile sizes. Problem sizes were the same as those used in Section 7.3. We did not apply tiling to CFFT2D because its imperfect nesting and use of indirection arrays would likely frustrate any form of automatic loop restructuring.

Figure 7.7 shows the results. Execution times are plotted against tile sizes. In each case, the performance for the largest tile size is effectively that of the untiled version of the same loop because the largest tile size equals the number of iterations[3], which means there is only one big tile for that loop. In some cases, tiling was not appropriate after loop or array restructuring because locality had improved so much that subsequent tiling brought no further benefit. To reflect this in the graphs, we show the execution time of the untiled loop as a horizontal line without individual data points for different tile sizes.

From Figure 7.7, we see that tiling improved the performance of the original loop substantially, as previous studies have also found [Lam, Rothberg, and Wolf 1991; Li 1995]. However, without loop or array restructuring, performance improvement depended very much on the tile size. The performance curves are generally U-shaped, as Figure 7.7(a)

2. We applied tiling by hand even though SUIF itself has implemented this technique because its current implementation is insufficient for our purpose. The compiler tiled none of the loops when loop bounds were runtime constants, possibly because the compiler decided against tiling when it could not ascertain whether the loops were large enough to justify it. With loop bounds set to large compile-time constants, three (MATMUL, MXM, and one pair of inner loops in CHOLSKY) of the eight loops were tiled automatically.

3. The only exception is SYR2K after loop restructuring. Its innermost loop has many times more iterations than that of the original version. If we were to show this iteration count as the largest tile size (as we do in all other cases), either the x-axis would be awkwardly long or the small performance variation with tile size would become illegible.



If tiling is not appropriate, execution times for untiled loops are shown as horizontal lines without data points.

CFFT2D is omitted because it cannot be tiled automatically.

Figure 7.7: Array Restructuring and Tiling

demonstrates most evidently. Execution time decreases with increasing tile size, reaches a minimum, and then rises again. This observation also agrees with previous findings [Lam, Rothberg, and Wolf 1991; Li 1995]. Some researchers have therefore investigated how to choose an appropriate tile size for best performance [Coleman and McKinley 1995; Esseghir 1993; Lam, Rothberg, and Wolf 1991].

As Figure 7.7 shows, array and loop restructuring sometimes made the performance of the tiled loops less sensitive to tile size. Performance still deteriorated when tiles were too small, presumably because of excessive loop overhead. However, performance degradation for large tiles was mitigated. In several cases, tiling brought no further improvement after arrays had been restructured; the "optimal" tile was the largest possible, namely all the iterations. Loop restructuring had a similar effect, as a previous study has also observed [Li 1995]. However, it is always valid to apply any array restructuring and tiling together. Since array restructuring does not alter the loop structure, it never makes an originally tilable loop untilable. The same cannot be said of loop restructuring in general.

7.5 Summary

In our experiments, array restructuring by itself improved performance significantly in most cases, with a modest, sometimes insignificant, runtime overhead. In other cases, however, the overhead was too high for the technique to be beneficial. The prototype compiler was able to distinguish between the two. We also found that array restructuring complemented loop restructuring: it applied to cases where the latter did not and performed comparably when both applied; when it did not perform as well, some simple loop restructuring would have sufficed. Moreover, array restructuring also complemented loop restructuring in that it made tiling performance less dependent on the size of tiles.

Chapter 8

Parallel Execution

This dissertation focuses on restructuring arrays to improve locality for loop execution on a single processor. However, array restructuring may also help parallel execution. At the very least, by improving the locality of each processor's access pattern, it can decrease the execution times on individual processors and thus the overall execution time as well. Moreover, changing array layouts may alleviate false sharing, a problem that arises from a mismatch between data layouts and inter-processor access patterns and has been successfully addressed by means of data transformations [Jeremiassen and Eggers 1995]. Therefore, we also carried out some experiments for parallel execution to gauge how our array restructuring technique affects performance in the parallel case. Results are discussed in this chapter.

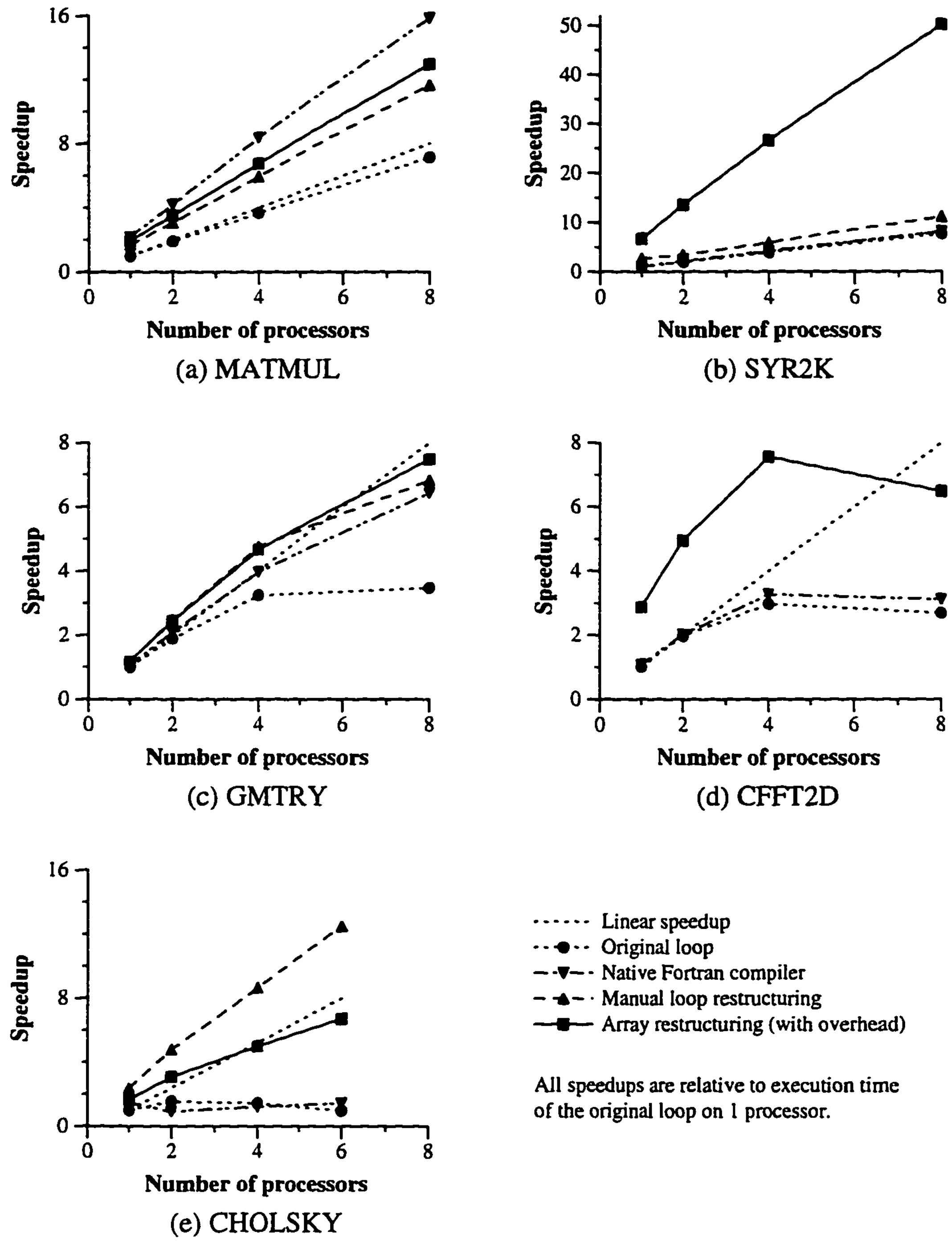
8.1 Experiments and Results

The experiments were performed on a Silicon Graphics Power Challenge [SGI 1995a]. (Results on a Kendall Square Research KSR-2 are included in Appendix B. They resemble the Power Challenge results here.) The Power Challenge is a shared-memory multiprocessor based on MIPS R8000 processors. Each processor has two levels of cache for integer data, but only one for floating point data. The first-level cache is an on-chip, 16 KB, write-through, direct-mapped cache with a line size of 32 bytes. However, it is used only for integer operations, while all the loops in our experiments perform primarily floating point operations. Floating point loads and stores are satisfied directly by a second-level cache: an off-chip, 4 MB, write-back, four-way set-associative cache with a line size of 512 bytes. The MIPS R8000 also has a 384-entry three-way set-associative translation lookaside buffer (TLB) shared by instructions and data. (All background information in this paragraph is from SGI's on-line documentation [SGI 1995a].)

As before, we first compiled the Fortran code using our prototype compiler, and then the output C code using Power Challenge's native C compiler (`cc`), both with standard compiler optimizations (`-O2`). SUIF automatically parallelized the loops where it identified such opportunities and inserted calls to its runtime system for parallel execution. The object code was finally linked with SUIF's and our runtime systems.

Figure 8.1 shows the results for loops that our prototype compiler considered profitable; Figure 8.2 shows those for the unprofitable cases with manual override of the compiler's decision against array restructuring. Each graph plots the parallel speedup against the number of processors for several cases. In addition to the usual categories explained in the previous chapter, all of which involve SUIF in some way, we also measured the loops directly parallelized and compiled by Power Challenge's native Fortran compiler. This allows us to further validate any trend that may be observed from the SUIF-based results. We do not, however, quantitatively compare the two sets of results because they come from two vastly different compilers, one being a commercial product and the other a research prototype.

All speedups were computed relative to the execution time of the parallelized version of the original loop (compiled by SUIF) running on a single processor. Therefore, for other versions of the same loop, the speedup may, and often does, exceed the number of processors because the speedup is in part due to reasons that are not directly related to parallel execution. Also, note that parallel speedups strictly speaking should be calculated based on the execution time of the sequential version, rather than a parallelized version running on one processor. However, our focus here is not parallelization overhead, but how array restructuring affects parallel execution. Using the sequential execution time as a basis would merely scale the vertical axes but not influence the relative trends.



All speedups are relative to execution time of the original loop on 1 processor.

Figure 8.1: Parallel Speedups on SGI Power Challenge — Profitable Cases

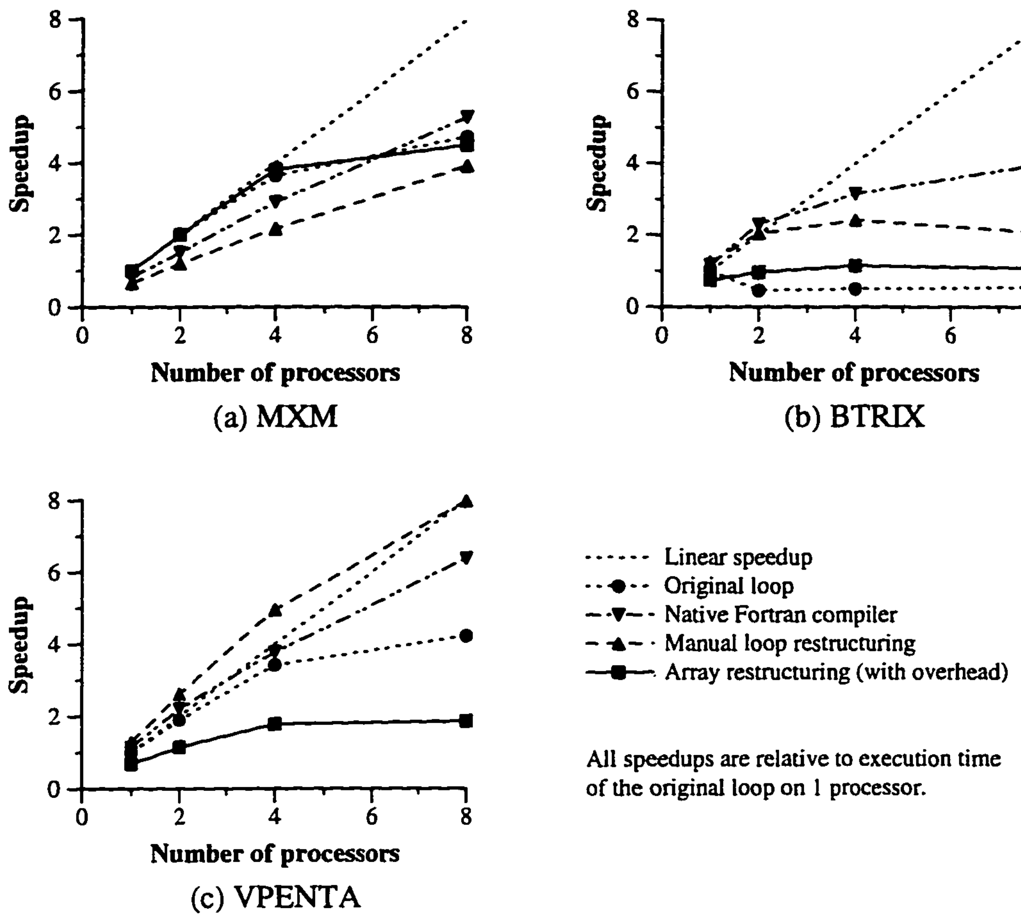


Figure 8.2: Parallel Speedups on SGI Power Challenge — Unprofitable Cases (with Manual Override of Compiler Decision)

8.2 Discussion

Let us first consider Figure 8.1, which shows the loops for which the prototype compiler considered array restructuring profitable. Array restructuring did substantially improve performance over the original loop. This agrees with our previous observation on uniprocessor execution. Array restructuring achieved this in two ways: improving the spatial locality of execution on each processor and reducing false sharing between processors.

Better spatial locality accounts for the performance improvement in MATMUL, SYS2K, and CFFT2D, as in the workstation results reported in the last chapter. MATMUL and SYS2K are both perfect loop nests that can be parallelized easily. In both, the outermost loop is parallel, and its iterations have little interaction. Thus, all the speedup curves are almost linear. Array restructuring improved performance by roughly the same factor on any number of processors. It was, incidentally, only applied to read-only arrays. The improvement could not have come from reduced false sharing. Instead, it resulted from better spatial locality of loop execution on each individual processor. Essentially, what we expect on a single processor occurred independently on each processor of the multiprocessor. As for CFFT2D, the situation was similar, except that the restructured array was both read and written, but still with no false sharing. Performance did not scale well with more processors because the compilers (SUIF as well as the native Fortran compiler) were able to parallelize only the innermost loop.

False sharing comes into play for GMTRY and CHOLSKY. For GMTRY, on one processor, performance for the four loop versions was comparable. On eight processors, however, the original loop performed much worse than the other three. This was due to false sharing. In GMTRY, the parallelized loop writes a column of elements (in a two-dimensional array), which are stored consecutively according to the Fortran convention. False sharing may result if different processors write different elements in the same cache line. Both loop and array restructuring can remedy this problem, as evidenced in Figure 8.1(c) by the comparable performance of all but the original loops. As in the workstation results, array and loop restructuring had similar performance. Array restructuring in effect transposes the array, causing the parallelized loop to write elements far apart, and a sequential loop within it to write consecutive elements. This both improves spatial locality and reduces false sharing. Loop restructuring (done manually as well as by the native Fortran compiler) can achieve a similar effect with loop distribution followed by interchange.

CHOLSKY suffered a similar problem, and array restructuring offered a similar solution. Figure 8.1(e) shows hardly any parallel speedup for the original loop. False sharing between processors, poor spatial locality on each individual processor, and load imbalance (since the parallelized loop had only a few iterations) all contributed to the problem. Array restructuring solved the first two, by allowing the outermost, parallelized loop to write elements far apart and the inner loops executed entirely on one processor to go through elements consecutively. A suitable sequence of loop transformations that effectively turns the imperfect loop nest "inside out" can solve all three problems. Thus, manual loop restructuring performed much better than array restructuring, unlike in the workstation's case where the two performed comparably. However, as discussed in Section 7.3.2, it is difficult to find the right loop transformation sequence. Failing to do that, the native Fortran compiler did not improve performance.

Finally, let us briefly look at the unprofitable cases in Figure 8.2. (Recall that in these cases array restructuring was applied against the prototype compiler's decision for experimental purposes.) Generally, the observations resemble those for single processor execution (see Section 7.3.2). For MXM, array restructuring had virtually no effect on performance, whereas loop restructuring, including that performed by the native Fortran compiler, decreased performance unexpectedly. However, loop restructuring did have an advantage at eight processors because of a better distribution of iterations among processors. For BTRIX and VPENTA, simple loop restructuring chosen by the native Fortran compiler obtained significant improvement over the original loops, whereas array restructuring performed dismally after copying overhead has been taken into account.

8.3 Summary

Although our array restructuring targets execution on a single processor, it may also improve performance of parallel execution. First, it improves the spatial locality of loop execution on each individual processor. Moreover, it may reduce false sharing between processors. Laying out an array in such a way that inner loops executed entirely on one processor go through the elements consecutively tend to separate the elements accessed by parallelized, outer loops by a large distance, thus decreasing the potential for false sharing. We could in principle use array restructuring to target false sharing more specifically because false sharing, like spatial locality, is related to the layout of data in memory. However, we have not pursued this direction further in this study since our focus is on sequential execution.

Parallel execution introduces more dimensions into the problem of program restructuring. In addition to temporal and spatial locality, both important on one processor, we also have to consider parallelism, inter-processor communication, and load balance, to name just a few. Loop restructuring may affect all these factors. It is difficult to find a suitable loop transformation to balance often conflicting goals, even harder than for single processor execution. This suggests that it is even more important to combine array and loop restructuring in parallel programs than in sequential ones.

Part IV

Conclusion

Chapter 9

Related Work

In this chapter, we review related work that also aims to improve cache performance. As discussed earlier, one may enhance the locality of array accesses in loops by restructuring loops or by restructuring arrays. This chapter is organized accordingly, but with an emphasis on array restructuring techniques since they are more closely related to this study.

9.1 Array Restructuring

Our survey on array restructuring techniques is divided into two parts. Since we are mainly concerned with changing the storage order of array elements, we first discuss in detail related work with a similar focus. Then, we also look at other forms of data layout transformations and optimizations.

9.1.1 Reordering Array Elements

We review three other studies of reordering array elements for better cache performance. These studies share several characteristics that differentiate them from our work. First, they use both loop and array restructuring; we focus on array restructuring. Second, they address cache performance on shared address space multiprocessors; our main concern is single processor execution. Third, they consider a small subset of possible array transformations, typically permutations of array dimensions; we explore a much larger design space. Fourth, they try to select a single transformation for each array maintained throughout multiple loop nests; we allow dynamic restructuring between loop nests. As we discuss each study, we elaborate on these and other differences.

Cierniak and Li have proposed a unified framework for loop and array restructuring [Cierniak and Li 1995]. It extends Li's previous loop restructuring framework based on nonsingular matrices [Li and Pingali 1993c]. Essentially, they observe that for a traditional row-major or column-major array layout, the memory location of an element is an affine function of its indices. (Recall that, as discussed in Chapter 4, the scalar offset of an element into the array is the dot product of the index and linearization vectors, plus some scalar constant.) They propose that a compiler can choose from all affine functions rather than always adhere to a standard layout. This flexibility, together with the freedom to change loop structures, offers many opportunities to improve locality.

Their array restructuring framework differs from ours in not having the notion of transformed index vectors. In their framework, the compiler chooses directly an affine function taking original array indices to memory locations; in ours, the compiler selects an index transformation that takes original index vectors to transformed index vectors, which in turn are used to compute memory locations. Any transformation in either framework can be expressed in the other. In this sense, both are equally general.

Although their framework is potentially very general, the problem of actually choosing the right combination of array layout and loop transformation is tractable only because many possibilities are ruled out *a priori*. Imposing several conditions on array layouts, they eventually consider only layouts that effectively correspond to permutations of array indices (in two dimensions, the possibilities reduce to row-major and column-major layouts). Also, instead of computing a loop transformation with an algorithm, as done in Li's earlier work [Li and Pingali 1992; Li and Pingali 1993a; Li 1995], they consider a predetermined subset of transformations that they believe to be sufficient. Given these restrictions, they select a combination of array and loop transformations by exhaustively evaluating $m!2^{n^2}$ combinations of candidates for legality and potential locality improvement (for an n-deep loop nest and m-dimensional array). As for multiple accesses, the compiler resolves conflicting loop and array transformation requirements from different accesses apparently also with an exhaustive search, although details were not given.

In contrast, our algorithm explores the full range of array transformations allowed by the framework, including index space "skewing" and transformations for partial restructuring. Moreover, we explore the possibilities without an exhaustive search, which would be impossible because the number of transformations to consider is infinite. For example, we have explained in Section 3.1.1 that for SYR2K, permuting array dimensions cannot solve the performance problem; the index space must be skewed. This possibility, though allowed in both frameworks, is explored and selected only by our algorithm. More generally, access patterns involving diagonal movement through the index space, often found in banded matrix computations, require similar array transformations.

The SUIF compiler restructures arrays to reduce false sharing and improve processor and spatial locality [Anderson, Amarasinghe, and Lam 1995]. The SUIF project aims at automatically parallelizing sequential programs. The compiler decides how to partition array elements and loop iterations across processors in a way that maximizes parallelogram while minimizing inter-processor communication [Anderson and Lam 1993]. For machines with a shared address space but distributed memories (such as Stanford's DASH [Lenoski et al. 1992]), one might rely on the cache coherence hardware to move data between processors as needed. However, this straightforward solution may lead to false sharing and poor processor and spatial locality. These problems arise because in the shared address space, array elements supposedly allocated to the same processor may be separated by elements belonging to other processors [Anderson, Amarasinghe, and Lam 1995].

To remedy the problem, Anderson et al. propose restructuring arrays based on their data partitions [Anderson, Amarasinghe, and Lam 1995]. Elements allocated to the same processor are put in a contiguous region in the shared address space; in effect, such contiguous regions serve as the "local memories" of processors. Arrays are restructured with two transformations: *permutation* permutes array dimensions; *strip-mining* turns one array dimension into two in much the same way as loop strip-mining turns a singly nested loop into a doubly nested one. Anderson et al. also describe a linear algebraic framework for array restructuring that resembles ours in using linear index transformations.

Their work differs from ours in several ways. First, they use their framework only for representing array permutations. (Strip-mining cannot be represented by linearly transforming index vectors.) They mentioned the possibility of unimodular transformations but did not pursue it. In comparison, we fully explore unimodular and even non-unimodular invertible transformations. Second, they restructure arrays solely to bring elements allocated to the same processor together in the shared address space. They are not concerned about the storage order of those elements, expecting another compilation phase to reorder elements for better cache performance. In contrast, the storage order of array elements on a single processor is precisely our focus. In this sense, our work complements theirs. Finally, the particular sets of transformations are also different; neither includes the other.

Ju and Dietz have proposed combining array and loop restructuring to reduce cache coherence overhead in shared-memory multiprocessors [Ju and Dietz 1992]. For each loop and each array the loop accesses, the compiler estimates the cache coherence overhead for all possible combinations of loop structures and data layouts. Calculated from a detailed machine-specific cost model, these estimates are used to construct an *interference graph*. Nodes of the graph represent loops and arrays; edges connect loops to the arrays they access and are labeled with a cost table containing the above estimates. The compiler selects loop structures and array layouts to minimize the sum of the edge costs, which is the expected total cache coherence overhead. Since all combinations of loop structures and data layouts are exhaustively searched (with some pruning), in practice the compiler can afford to consider only a small number of possibilities [Ju and Dietz 1992]. In the only example given, arrays are either row-major or column-major, and loops are interchanged.

This work differs from ours in several ways. First, the goals are different: it aims to reduce cache coherence traffic on shared-memory multiprocessors, while we seek to improve spatial locality on a single processor. Second, for array restructuring, it can afford to consider only a few possible transformations (e.g., row- and column-major storage orders), while our framework allows efficient exploration of a much larger design space. Third, they use both loop and array restructuring, while we focus on the latter. Finally, they analyze loops and arrays on a global level to select one layout for each array, while we analyze individual loops locally, restructuring arrays in between if necessary.

9.1.2 Other Forms of Data Layout Optimizations

The following optimizations differ from our array restructuring technique fundamentally in that they do not focus on reordering array elements as we do.

Array padding has long been used to improve memory system performance [Bacon, Graham, and Sharp 1994]. The compiler adds dummy elements to an array so that elements accessed consecutively are separated by some suitable distance in memory. This technique is useful for fully utilizing the memory bandwidth on vector machines with banked memory: the padding causes consecutive memory accesses to be directed to different memory banks and thus serviced in a pipelined fashion. Array padding may also be needed to avoid excessive conflict misses in caches with low associativities. In such caches, excessive conflict misses may result if the program happens to access array elements at strides that equal powers of two (which often happens when some array dimensions are powers of two) because all those elements would be mapped to a small number of sets in the cache. Array padding solves the problem by slightly shifting those elements so that they are mapped to different sets. In a related technique, *array alignment*, the compiler judiciously chooses array starting addresses to avoid the conflict misses that may occur if the program alternately accesses elements of different arrays and those elements happen to be mapped to the same set in the cache. The systematic selection of suitable padding amounts has been studied recently [Bacon et al. 1994].

Array padding is orthogonal to our array restructuring technique. Unlike our technique, it does not reorder array elements, just adds dummy elements between the "real" elements. However, it can be incorporated into our framework. When calculating the linearization vector, we can account for the dummy elements by choosing the vector's components to be slightly greater than what would be strictly required for all elements to have their own unique memory locations.

Jeremiassen and Eggers apply data transformations to reduce false sharing in coarse-grained, explicitly parallel applications on shared-memory multiprocessors [Jeremiassen and Eggers 1995; Jeremiassen 1995]. Based on how different threads of execution access data, the compiler transforms data structures so that data accessed by the same thread are grouped together while write-shared data with little processor locality are separated into different cache lines. Three data transformations are used: *group and transpose* converts a group of statically declared vectors into a vector of records with one field for each original vector; *pad and align* pads data structures to the cache line size and aligns them on cache line boundaries; *indirection* places dynamically allocated data structures predominantly written by one processor in a heap region specific to that processor, leaving a pointer to the new location in the original location [Eggers and Jeremiassen 1991; Jeremiassen 1995; Jeremiassen and Eggers 1995].

Their work and ours differ in several significant ways. First, they aim at reducing false sharing in explicitly parallel applications on shared-memory machines, while we try to improve spatial locality in sequential programs running on uniprocessors. Second, their transformations target data structures in general, while ours focus on reordering array elements. In particular, as applied to multidimensional arrays, group and transpose amounts to a form of permuting array dimensions: if the vector components are themselves arrays, this transformation effectively permutes the array dimension corresponding to the vector to the leftmost index position while preserving the relative order of the other dimensions. Third, their transformations consist of individual useful techniques selected heuristically by the compiler, while our transformations are represented in a formal framework and chosen by algorithmic analysis of formal representations. Finally, they make data restructuring decisions after a static whole-program analysis, while we may dynamically restructure arrays between loops and uses local analysis.

Torrellas, Lam, and Hennessy have also studied the impact of false sharing and spatial locality in multiprocessor caches [Torrellas, Lam, and Hennessy 1994]. They propose five data layout optimizations to improve cache performance. These include different forms of padding and alignment, dynamically allocating data structures in processor-specific heaps, and putting locks and the variables they protect together. Their techniques generally speaking differ from ours in the same ways as those of Jeremiassen and Eggers just discussed.

9.2 Loop Restructuring

In this section, we review loop restructuring techniques for improving locality — both temporal and spatial — to demonstrate the depth and breadth of the long-standing work in this area. In particular, we discuss loop interchange, permutation, fusion, and tiling (or blocking) [Bacon, Graham, and Sharp 1994]. In addition, we survey other transformations that do not improve locality by themselves but are sometimes needed to enable those just listed, as well as frameworks for systematically selecting sequences of transformations. In keeping with our theme, we focus on how the transformations enhance locality, although many of them may also serve other purposes such as increasing instruction-level parallelism, reducing loop overhead, and enabling automatic parallelization or vectorization.

Loop interchange swaps two different loops in a loop nest, one of which is typically the innermost loop [Allen and Kennedy 1984]. This transformation reorders iterations so that those accessing the same or nearby data are executed closer together in time than under the original program order. It can improve temporal locality by, say, moving a loop that reuses the same array element to the innermost position, or spatial locality by similarly moving a loop that goes through an array at unit stride.

Loop permutation generalizes loop interchange. It permutes the loops in a loop nest into some order different from the original program order. A permutation may be viewed as a series of interchanges. For loop interchange and permutation to be legal, all the transformed loop nest's dependence vectors, obtained by permuting components of the original dependence vectors accordingly, must remain lexicographically positive. Some researchers have studied selecting a permutation for both parallelism and locality based on a simple memory model [Carr, McKinley, and Tseng 1994; Kennedy and McKinley 1992].

Loop skewing and *loop reversal* by themselves do not improve locality but are often required to make loop permutation legal [Wolfe 1989b]. Loop skewing merely skews the iteration space by adding some multiple of an outer loop variable to the bounds of inner loops; by itself, it does not reorder iterations and thus is always legal. Loop reversal does reorder iterations, reversing the order in which a loop goes through its iterations [Wedel 1975]. Therefore, loop reversal is legal only if all dependence vectors remain lexicographically positive after their components for the reversed loop have been negated [Bacon, Graham, and Sharp 1994].

Given a loop nest, it is often difficult to choose a legal sequence of such loop transformations to achieve the desired effects on locality. The difficulty is compounded by the fact that a sequence may be legal overall even if some intermediate steps are illegal. Some researchers have therefore proposed representing these transformations in a formal framework amenable to mathematical analysis [Banerjee 1991; Wolf and Lam 1991b]. Specifically, these loop transformations and their combinations are represented by unimodular transformations on the iteration space. A compiler represents array accesses and loop bounds as affine functions, selects a unimodular transformation mathematically, computes the transformed affine functions, and generates code directly from these transformed functions — all without explicitly applying any of the intermediate loop transformations. This mathematics-based approach has inspired other research efforts, including, of course, this dissertation. Others have also developed a framework to represent iteration reordering loop transformations as sequences of kernel transformations each with specific rules for converting dependence vectors, loop bounds, etc. [Sarkar and Thekkath 1992]. It facilitates legality checking and code generation for a given transformation sequence, although algorithms must still be developed to choose a suitable sequence.

The unimodular loop restructuring framework has notably been extended to one using nonsingular matrices, which subsume unimodular matrices [Li and Pingali 1993c]. This extension adds another loop transformation called *loop scaling*, which replaces a loop variable by a multiple of itself [Li 1993]. This framework has led to *access normalization* [Li and Pingali 1992; Li and Pingali 1993a]. Access normalization is an optimization for non-uniform memory access (NUMA) multiprocessor. Given a distribution of data to processors, it aims to transform a loop nest so that the outermost loop can be parallelized without compromising processor locality, while in the inner loops, data can be transferred between processors using block transfer memory operations [Li 1993]. The same framework has also led to cache locality optimization techniques especially suited for banded matrix applications on uniprocessors [Li 1995; Li 1993]. As discussed in Section 9.1.1, some recent work on unifying loop and array restructuring has also built on this framework [Cierniak and Li 1995].

The above optimizations target perfect loop nests because they reorder iterations as a whole rather than statements in an iteration. To apply them to imperfect loop nests, a compiler may use *loop distribution* to convert the original loop nest into a series of perfect loop nests, each containing some of the statements in the original loop body [Kennedy and McKinley 1990]. Even when starting with a perfect loop nest, we may still want to distribute it if loop transformations that will improve the performance of some statements in the loop body are precluded by loop-carried dependences arising from others.

Loop fusion, the inverse of loop distribution, can also improve locality [Abu-Sufah 1979; Wolfe 1989b]. By fusing corresponding iterations from different loop nests into one, it brings uses of the same or nearby data, originally in corresponding iterations separated by an entire loop nest, to within the same iteration. It has been used together with loop distribution to maximize loop-level parallelism and improve locality on shared-memory multiprocessors [Kennedy and McKinley 1994], and furthermore with loop permutation and reversal to improve locality on uniprocessors as well [Carr, McKinley, and Tseng 1994].

Finally, *loop tiling* (or *blocking*) is another powerful iteration reordering technique to improve locality [Abu-Sufah 1979; Gannon, Jalby, and Gallivan 1988]. Tiling can be viewed as strip-mining followed by permutation [Wolfe 1989a]. The iteration space is divided into rectangular "tiles" such that the data touched by a tile fit in the cache. The restructured loop executes iterations one tile at a time, finishing one tile before executing another. Thus, each tile reuses data in the cache as much as possible before the next tile replaces them with other data.

Loop tiling has long been used by scientific programmers in the form of blocked algorithms [Lam, Rothberg, and Wolf 1991]. Compiler algorithms to automate tiling is relatively recent [Wolf and Lam 1991a]. The compiler applies loop permutation, skewing, and reversal within a unimodular transformation framework as discussed above to produce a *fully permutable nest* — a nest of loops for which any permutation is legal — which can then be tiled [Wolf and Lam 1991a; Wolf 1992]. Others have looked into the possibility of using non-rectangular tiles [Irigoin and Triolet 1988].

However, tiling performance is often sensitive to tile size [Lam, Rothberg, and Wolf 1991]. If a tile is too large, the data it touches may overflow the cache, destroying the intra-tile reuse critical to tiling's potential to improve performance. This is especially true for caches with low associativity, in particular direct-mapped caches, because conflict misses may occur even if the data size is well within the cache's capacity. Proposed solutions to this problem include copying data accessed by each tile to a separate buffer while they are being used [Lam, Rothberg, and Wolf 1991], judiciously (and conservatively) choosing tile sizes with compiler analysis [Coleman and McKinley 1995], and combining tiling with other forms of loop restructuring [Li 1995].

Chapter 10

Summary and Conclusions

This dissertation studies the use of array restructuring to improve the locality of array accesses in loops. Under this approach, the compiler analyzes the access pattern for each array to determine how elements should be laid out for better locality. We explore how this can be done effectively — efficiently, automatically, and as generally as possible.

To this end, we have developed a framework for array restructuring. In this framework, a restructured array stores the same elements as the original, but in a different order defined by a linear transformation of array index vectors. Using linear transformation is efficient: it incurs no extra indexing overhead, even in the case of non-affine array index expressions. Moreover, it makes possible automatic analysis based on linear algebraic techniques. Finally, it is extremely general: for example, it subsumes permuting array dimensions and allows restructuring only regularly spaced elements that are actually accessed by the loop.

Within this framework, we have devised algorithms to automate array restructuring.

- First, the compiler chooses an index transformation for each array based on the access pattern so that, first and foremost, the innermost loop accesses elements consecutively. Among other things, we can handle non-affine index expressions and imperfect loop nests, both frequent obstacles to loop restructuring.
- Second, the restructured array is linearized: its elements are laid out in such a way that an element's location is an affine function of the array indices. Though easy for traditional, constant array bounds, linearization is complicated by the non-constant bounds that may result from general array restructuring. Our algorithm offers a gen-

eral solution that guarantees low indexing overhead at the price of some unused memory.

- Finally, we can also restructure only those elements of an array that are really accessed by the loop, thus reducing memory use and copying overhead as well as improving spatial locality by storing elements more compactly. Our techniques support subsets of elements that are regularly spaced or within a restricted range, or both, in the index space.

We obtained promising results from experiments using loops from related literature. We found that for a range of problem sizes, array restructuring improved performance substantially in many cases, despite a modest overhead in some under pessimistic assumptions. Our prototype was able to identify those cases where array restructuring was unprofitable due to excessive overhead.

We also found that array restructuring compared favorably with loop restructuring in both applicability and performance. The two approaches complement each other; neither enjoys an unqualified superiority. Array restructuring applied in some cases where loop restructuring did not and achieved comparable, sometimes better, performance when both were applicable. However, there were also cases where simple loop restructuring outperformed our array restructuring technique — precisely those cases for which our prototype decided against array restructuring because it might not be profitable.

These quantitative results confirm the qualitative observation at the beginning of this dissertation that loop and array restructuring each has its advantages and disadvantages, which often mirror each other.

- Array restructuring can be more easily applied to complicated loop structures. For example, we have discussed how it handles imperfect loop nests, non-affine index expressions, etc., with relative ease. Unlike loop restructuring, it is also immune to insufficient or imprecise information on loop-carried dependences.

- Array restructuring changes spatial locality but not temporal locality. Thus, it can improve the former without jeopardizing the latter. Loop restructuring changes both, which makes it more powerful (it can improve both) but also complicates performance tradeoffs.
- Array restructuring can improve independently the locality for each array accessed in a loop nest, while loop restructuring affects accesses to all arrays in the loop, thus necessitating tradeoffs between them. On the other hand, multiple loop nests can be restructured independently, while array restructuring affects all loop nests accessing the same array. While this latter case may occur more often than the former, they are not mutually exclusive. In the latter case, dynamically restructuring arrays between loop nests may help, though at the expense of runtime overhead.

The complementary nature of loop and array restructuring suggests that the best results are likely to be obtained by integrating both approaches. However, simply performing them one after the other in some order may not produce the expected performance advantage. There have been attempts to fully integrate the selection of loop and array transformations. As noted in the discussion on related work, in these previous attempts the problem of choosing the right combination is tractable only because many possible transformations — loop or array — are ruled out *a priori*. The potential of integrating the widest possible classes of loop and array transformations may not be fully realized.

This dissertation is less ambitious in that it focuses on array restructuring. Loop restructuring has been widely studied for a long time, with the tremendous amount of research effort culminating in the state of the art. Array restructuring has so far received relatively little attention. It is hoped that this study has contributed to a deeper understanding of effective array restructuring. In the short term, the techniques developed can be used on their own as another set of tools available to an optimizing compiler for improving locality. In the longer term, the insights obtained may also serve as one small step on the road to an eventual integration of both approaches.

Bibliography

[Abu-Sufah 1979] W. Abu-Sufah. "Improving the Performance of Virtual Memory Computers." Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1979.

[Aho, Sethi, and Ullman 1986] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Reading, MA: Addison-Wesley, 1986.

[Allen and Kennedy 1984] J. R. Allen and K. Kennedy. "Automatic Loop Interchange." In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction,* (Montreal, Canada, June), 233-46. New York: ACM Press, 1984.

[Alpern, Wegman, and Zadeck 1988] B. Alpern, M. N. Wegman, and F. Zadeck. "Detecting Equality of Values in Programs." In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages,* (San Diego, CA, January), 1-11. New York: ACM Press, 1988.

[Amarasinghe and Lam 1993] Saman P. Amarasinghe and Monica S. Lam. "Communication Optimization and Code Generation for Distributed Memory Machines." In *Proceedings of ACM SIGPLAN '93 Conference on Programming Language Design and Implementation,* (Albuquerque, NM, June), 126-38. New York: ACM Press, 1993.

[Anderson and Lam 1993] Jennifer M. Anderson and Monica S. Lam. "Global Optimizations for Parallelism and Locality on Scalable Parallel Machines." In *Proceedings of ACM SIGPLAN '93 Conference on Programming Language Design and Implementation,* (Albuquerque, NM, June), 112-25. New York: ACM Press, 1993.

[Anderson, Amarasinghe, and Lam 1995] Jennifer M. Anderson, Saman P. Amarasinghe, and Monica S. Lam. "Data and Computation Transformations for Multiprocessors." In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* (Santa Barbara, CA, July), 166-78. New York: ACM Press, 1995.

[Bacon, Graham, and Sharp 1994] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. "Compiler Transformations for High-Performance Computing." *ACM Computing Surveys* 26(4):345-420 (December 1994).

[Bacon et al. 1994] David. F. Bacon et al. "A Compiler Framework for Restructuring Data Declarations to Enhance Cache and TLB Effectiveness." In *Proceedings of CASCON '94,* (Toronto, Canada, November), 270-82. Ottawa: National Research Council of Canada, 1994.

[Bailey and Barton 1986] David H. Bailey and John T. Barton. "The NAS Kernel Benchmark Program." Numerical Aerodynamic Simulations Systems Division, NASA Ames Research Center, 1986. Distributed electronically with code. University of Illinois at Urbana-Champaign, 1979.

[Banerjee 1991] Utpal Banerjee. "Unimodular Transformations of Double Loops." In *Advances in Languages and Compilers for Parallel Processing,* edited by A. Nicolau et al. Research Monographs in Parallel and Distributed Computing. Cambridge, MA: MIT Press, 1991.

[Bannon and Keller 1995] Peter Bannon and Jim Keller. "Internal Architecture of Alpha 21164 Microprocessor." In *Digest of Papers, COMPCON Spring '95, Technologies for the Information Superhighway,* 79-87. Los Alamitos, CA: IEEE Computer Society Press, 1995.

[Bloom 1979] David M. Bloom. *Linear Algebra and Geometry.* Cambridge, England: Cambridge University Press, 1979.

[Carr, McKinley, and Tseng 1994] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. "Compiler Optimizations for Improving Data Locality." In *Proceedings of Sixth International Conference on Architectural Support for Programming Languages and Operating Systems,* (San Jose, CA, October), 252-62. New York: ACM Press, 1994.

[Ciemiak and Li 1995] Michal Ciemiak and Wei Li. "Unifying Data and Control Transformations for Distributed Shared-Memory Machines." In *Proceedings of ACM SIGPLAN '95 Conference on Programming Language Design and Implementation, (*La Jolla, CA, June), 205-17. New York: ACM Press, 1995.

[Coleman and McKinley 1995] Stephanie Coleman and Kathryn S. McKinley. "Tile Size Selection Using Cache Organization and Data Layout." In *Proceedings of ACM SIGPLAN '95 Conference on Programming Language Design and Implementation,* (La Jolla, CA, June), 279-90. New York: ACM Press, 1995.

[DEC 1992] Digital Equipment Corporation (DEC). *DEC 3000 Model 400/400S AXP Technical Summary.* Maynard, MA: Digital Equipment Corporation, 1992.

[Dixit 1992] Kaivalya M. Dixit. "New CPU Benchmarks Suites from SPEC." In *Digest of Papers, COMPCON Spring '92, Thirty-Seventh IEEE Computer Society International Conference,* (San Francisco, CA, February), 305-10. Los Alamitos, CA: IEEE Computer Society Press, 1992.

[Dongarra et al. 1990] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. “A Set of Level 3 Basic Linear Algebra Subprograms.” *ACM Transactions on Mathematical Software* 16(1):1-17 (March 1990).

[Dutton et al. 1991] Todd A. Dutton, Daniel Eiref, Hugh R. Kurth, James J. Reisert, and Robin L. Stewart. “The Design of the DEC 3000 AXP Systems, Two High-Performance Workstations.” *Digital Technical Journal* 4(4):66-81 (Special issue 1992).

[Eggers and Jeremiassen 1991] Susan J. Eggers and Tor E. Jeremiassen. “Eliminating False Sharing.” In *Proceedings of the 1991 International Conference on Parallel Processing,* vol. 1, 377-81. University Park, PA: Pennsylvania State University Press, 1991.

[Esseghir 1993] K. Esseghir. “Improving Data Locality for Caches.” Master’s thesis, Rice University, 1993.

[Gannon, Jalby, and Gallivan 1988] Dennis Gannon, William Jalby, and Kyle Gallivan. “Strategies for Cache and Local Memory Management by Global Program Transformation.” *Journal of Parallel and Distributed Computing* 5(5):587-616 (October 1988).

[Hunt 1995] Doug Hunt. “Advanced Performance Features of the 64-Bit PA-8000.” In *Digest of Papers, COMPCON Spring '95, Technologies for the Information Superhighway,* 123-8. Los Alamitos, CA: IEEE Computer Society Press, 1995.

[Irigoin and Triolet 1988] F. Irigoin and R. Triolet. “Supernode Partitioning.” In *Proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Princi-*

ples of Programming Languages, (San Diego, CA, January), 319-29. New York: ACM Press, 1988.

[Jeremiassen 1995] Tor E. Jeremiassen. "Using Compile-Time Analysis and Transformations to Reduce False Sharing on Shared Memory Multiprocessors." Ph.D. dissertation, University of Washington, 1995.

[Jeremiassen and Eggers 1995] Tor E. Jeremiassen and Susan J. Eggers. "Reducing False Sharing on Shared Memory Multiprocessors through Compile Time Data Transformations." In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* (Santa Barbara, CA, July), 179-88. New York: ACM Press, 1995.

[Ju and Dietz 1992] Y.-J. Ju and H. Dietz. "Reduction of Cache Coherence Overhead by Compiler Data Layout and Loop Transformation." In *Languages and Compilers for Parallel Computing: Fourth International Workshop Proceedings, Ithaca, NY, USA,* edited by Utpal Banerjee et al., 344-58. Berlin: Springer-Verlag, 1992.

[Kelly and Pugh 1995] Wayne Kelly and William Pugh. "Finding Legal Reordering Transformations Using Mappings," In *Languages and Compilers for Parallel Computing: 7th International Workshop, Ithaca, NY, USA,* edited by Keshav Pingali et al., 107-24. Berlin: Springer-Verlag, 1995.

[Kennedy and McKinley 1990] Ken Kennedy and Kathryn S. McKinley. "Loop Distribution with Arbitrary Control Flow." In *Proceedings of Supercomputing '90,* (New York, NY, November), 407-16. New York: ACM Press, 1990.

[Kennedy and McKinley 1992] Ken Kennedy and Kathryn S. McKinley. "Optimizing for Parallelism and Data Locality." In *Proceedings of 1992 International Conference on Supercomputing,* (Washington, D.C., July), 323-34. New York: ACM Press, 1992.

[Kennedy and McKinley 1994] Ken Kennedy and Kathryn S. McKinley. "Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution." In *Languages and Compilers for Parallel Computing: 6th International Workshop Proceedings, Portland, OR, USA,* edited by Utpal Banerjee et al., 301-20. Berlin: Springer-Verlag, 1994.

[KSR 1992] Kendall Square Research Corporation (KSR). "Memory Management." In *Principles of Operation.* Revision 6.0. Waltham, MA: Kendall Square Research Corporation, 1992.

[KSR 1993] Kendall Square Research Corporation (KSR). "KSR/Series Memory Model and Processor." In *KSR/Series Parallel Programming.* Release dated 12/15/93. Waltham, MA: Kendall Square Research Corporation, 1993.

[Lam, Rothberg, and Wolf 1991] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. "The Cache Performance and Optimizations of Blocked Algorithms." In *Proceedings of Fourth International Conference on Architectural Support for Programming Languages and Operating Systems,* (Santa Clara, CA, April), 63-74. New York: ACM Press, 1991.

[Lenoski et al. 1992] D. Lenoski, J. Laudon, K. Charachorloo, W. D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. "The Stanford DASH Multiprocessor." *Computer* 25(3):63-79 (March 1992).

[Levitan, Thomas, and Tu 1995] David Levitan, Thomas Thomas, and Paul Tu. "The PowerPC 620 Microprocessor: A High Performance Superscalar RISC Microprocessor." In *Digest of Papers, COMPCON Spring '95, Technologies for the Information Superhighway,* 285-91. Los Alamitos, CA: IEEE Computer Society Press, 1995.

[Li 1993] Wei Li. "Compiling for NUMA Parallel Machines." Ph.D. dissertation, Cornell University, 1993.

[Li 1995] Wei Li. "Compiler Cache Optimizations for Banded Matrix Problems." In *Proceedings of 9th ACM International Conference on Supercomputing.* 1995.

[Li and Pingali 1992] Wei Li and Keshav Pingali. "Access Normalization: Loop Restructuring for NUMA Compilers." In *Proceedings of Fifth International Conference on Architectural Support for Programming Languages and Operating Systems,* (Boston, MA, October), 285-95. New York: ACM Press, 1992.

[Li and Pingali 1993a] Wei Li and Keshav Pingali. "Access Normalization: Loop Restructuring for NUMA Computers." *ACM Transactions on Computer Systems* 11(4):353-75 (November 1993).

[Li and Pingali 1993b] Wei Li and Keshav Pingali. "Loop Transformations for NUMA Machines." *SIGPLAN Notices* 28(1):9-12 (January 1993).

[Li and Pingali 1993c] Wei Li and Keshav Pingali. "A Singular Loop Transformation Framework Based on Non-Singular Matrices." In *Languages and Compilers for Parallel Computing: 5th International Workshop Proceedings, New Haven CT, USA,* edited by Utpal Banerjee et al., 391-405. Berlin: Springer-Verlag, 1993.

[Moore 1993] Charles R. Moore. "The PowerPC 601 Microprocessor." In *Digest of Papers, COMPCON Spring '93, Thirty-Eighth IEEE Computer Society International Conference*, (San Francisco, CA, February), 109-16. Los Alamitos, CA: IEEE Computer Society Press, 1993.

[Paap and Silha 1993] George Paap and Ed Silha. "PowerPC: A Performance Architecture." In *Digest of Papers, COMPCON Spring '93, Thirty-Eighth IEEE Computer Society International Conference*, (San Francisco, CA, February), 109-16. Los Alamitos, CA: IEEE Computer Society Press, 1993.

[Papworth 1996] David B. Papworth. "Tuning the Pentium Pro Microarchitecture." *IEEE Micro* 16(2):8-15 (April 1996).

[Pugh 1991] William Pugh. "Uniform techniques for Loop Optimization." In *Proceedings of the 1991 ACM International Conference on Supercomputing*, 341-52. 1991.

[Reif and Lewis 1986] J. H. Reif and H. R. Lewis. "Efficient Symbolic Analysis of Programs." *Journal of Computer and Systems Sciences* 32(3):280-313 (June 1986).

[Sarkar and Thekkath 1992] Vivek Sarkar and Radhika Thekkath. "A General Framework for Iteration-Reordering Transformations." In *Proceedings of ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, (San Francisco, CA, June), 175-87. New York: ACM Press, 1992.

[Schrijver 1986] Alexander Schrijver. *Theory of Linear and Integer Programming*. Series in Discrete Mathematics. New York: Wiley, 1986.

[SGI 1995a] Silicon Graphics, Inc. (SGI). "The POWER CHALLENGE Technical Report." Mountain View, CA: Silicon Graphics, Inc. 1995. Published as world-wide web page at http://www.sgi.com/Products/hardware/Power/brief-toc.html.

[SGI 1995b] Silicon Graphics, Inc. (SGI). *POWER Fortran Accelerator User's Guide.* Mountain View, CA: Silicon Graphics, Inc., 1995.

[Sites 1992] Richard L. Sites, ed. *Alpha Architecture Reference Manual.* Bedford, MA: Digital Press, 1992.

[Taylor 1992] Angus E. Taylor. "Partial Differentiation." In *McGraw-Hill Encyclopedia of Science and Technology.* 7th ed. Vol. 13. New York, 1992.

[Torrellas, Lam, and Hennessy 1994] Josep Torrellas, Monica S. Lam, and John L. Hennessy. "False Sharing and Spatial Locality in Multiprocessor Caches." *IEEE Transactions on Computers* 43(6):651-63 (June 1994).

[Tremblay and O'Conner 1996] Marc Tremblay and J. Michael O'Conner. "UltraSPARC-I: A Four-Issue Processor Supporting Multimedia." *IEEE Micro* 16(2):42-50 (April 1996).

[Wedel 1975] D. Wedel. "FORTRAN for the Texas Instrument ASC System." *SIGPLAN Notices* 10(3):119-32.

[Wilson et al. 1994] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S. W. Liao, C. W. Tseng, M. W. Hall, M. S. Lam and J. L. Hennessy. "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers." *SIGPLAN Notices* 29(12):31-37 (December 1994).

[Wolf 1992] Michael E. Wolf. "Improving Locality and Parallelism in Nested Loops." Ph.D. dissertation, Stanford University, 1992.

[Wolf and Lam 1991a] Michael E. Wolf and Monica S. Lam. "A Data Locality Optimizing Algorithm." In *Proceedings of ACM SIGPLAN '91 Conference on Programming Language Design and Implementation,* (Toronto, Canada, June), 30-44. New York: ACM Press, 1991.

[Wolf and Lam 1991b] Michael E. Wolf and Monica S. Lam. "A Loop Transformation Theory and an Algorithm to Maximize Parallelism." *IEEE Transactions on Parallel and Distributed Systems* 2(4):452-71 (October 1991).

[Wolfe 1989a] Michael J. Wolfe. "More Iteration Space Tiling." In *Proceedings of Supercomputing '89,* (Reno, NV, November), 655-64. New York: ACM Press, 1989.

[Wolfe 1989b] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers.* Research Monographs in Parallel and Distributed Computing. Cambridge, MA: MIT Press, 1989.

[Yeager 1996] Kenneth C. Yeager. "The MIPS R10000 Superscalar Microprocessor." *IEEE Micro* 16(2):28-41 (April 1996).

Appendix A

Memory Utilization

In this appendix, we show that the amount of memory needed for the restructured array (to facilitate address computation from array indices as described in Chapter 4) is at most $m!$ times the amount of memory for storing data elements, where m is the number of array dimensions. This upper bound is only approximate because we do not actually count elements, but only compare volumes of regions in the transformed index space that represent memory use. We believe, however, that the approximation is acceptable unless one or more array dimensions is unusually small.

Recall that in Chapter 4 we transform the original array bounds with a some index transformation to get the bounds of the restructured array. The transformed bounds define the set of elements that the restructured array must contain. These bounds are represented geometrically by the image polyhedron. Thus, in the following discussion we approximate the memory needed for array data with the volume of the image polyhedron. Recall also that to facilitate addressing an element, we allocate memory according to the relaxed bounds — a special form of bounds computed from the transformed bounds. The relaxed bounds are represented geometrically by the enclosing parallelepiped. Thus, we use the enclosing parallelepiped's volume to approximate the amount of allocated memory.

We show below that the volume of the enclosing parallelepiped is at most m! times that of the image polyhedron, assuming that the enclosing parallelepiped is in the form specified in Section 4.4.1 on page 75 and the image polyhedron is symmetric as defined in Section 4.5.2 on page 85. In our proof, we compute and compare the volumes of two regions: the enclosing parallelepiped itself and a polyhedron guaranteed to be inside the image polyhedron. The construction of the latter is discussed shortly in Section A.2, but first we prove several lemmas that will be used later. All summations and products are from 1 to m unless stated otherwise; the range is omitted to simplify notations.

A.1 Lemmas

Lemma A.1: For any vector γ and any m vectors P_k ($1 \le k \le m$), let P'_k be the mirror image of P_k about γ (i.e., $P'_k = 2\gamma - P_k$). The convex hull spanned by all $2m$ vectors P_k and P'_k (for $1 \le k \le m$) is equal to the set

$$\{\gamma + \sum \delta_k (P_k - \gamma) \ : \ \sum |\delta_k| \le 1\} \tag{A.1}$$

Proof: Let S' be the set in (A.1) and S be convex hull. By definition,

$$S = \{\sum (\alpha_k P_k + \beta_k P'_k) \ : \ \alpha_k \ge 0, \beta_k \ge 0, \text{and} \sum (\alpha_k + \beta_k) = 1\} \tag{A.2}$$

First, we show $S \subseteq S'$. Consider an element of S, denoted σ. Choosing $\delta_k = \alpha_k - \beta_k$, we can express σ as follows:

$$\begin{aligned}
\sigma &= \sum [\alpha_k P_k + \beta_k P_k'] \\
&= \sum [\alpha_k P_k + \beta_k (2\gamma - P_k)] \\
&= \sum [\alpha_k (P_k - \gamma) + \beta_k (\gamma - P_k)] + \sum (\alpha_k + \beta_k) \gamma \\
&= \sum [(\alpha_k - \beta_k)(P_k - \gamma)] + \gamma \\
&= \gamma + \sum \delta_k (P_k - \gamma)
\end{aligned} \tag{A.3}$$

Moreover, noting that all α_k and β_k are nonnegative and their sum is 1, we have

$$\sum |\delta_k| = \sum |\alpha_k - \beta_k| \le \sum (|\alpha_k| + |\beta_k|) = \sum (\alpha_k + \beta_k) = 1 \tag{A.4}$$

Therefore, $\sigma \in S'$ and hence $S \subseteq S'$

Conversely, consider an element of S', denoted σ'. We choose α_k and β_k from δ_k according to the following table:

	if $\delta_k \geq 0$	if $\delta_k < 0$
$1 \leq k < m$	$\alpha_k = \lvert\delta_k\rvert$ $\beta_k = 0$	$\alpha_k = 0$ $\beta_k = \lvert\delta_k\rvert$
m	$\alpha_m = \frac{1 - \sum\lvert\delta_k\rvert}{2} + \lvert\delta_m\rvert$ $\beta_m = \frac{1 - \sum\lvert\delta_k\rvert}{2}$	$\alpha_m = \frac{1 - \sum\lvert\delta_k\rvert}{2}$ $\beta_m = \frac{1 - \sum\lvert\delta_k\rvert}{2} + \lvert\delta_m\rvert$

It is clear that in all cases $\alpha_k \geq 0$ and $\beta_k \geq 0$. Furthermore, $\alpha_k + \beta_k = |\delta_k|$ for $1 \leq k < m$ and $\alpha_m + \beta_m = (1 - \sum|\delta_k|) + |\delta_m|$. Therefore,

$$\sum_{k=1}^{m} (\alpha_k + \beta_k) = \sum_{k=1}^{m-1} |\delta_k| + \left(1 - \sum_{k=1}^{m} |\delta_k|\right) + |\delta_m| = 1 \tag{1.5}$$

Hence, σ' belongs to S and therefore $S' \subseteq S$. Since S and S' are subsets of each other, they are equal.

Lemma A.2: For any nonnegative number λ and positive integer m,

$$\int_{|y_1| + \ldots + |y_m| \leq \lambda} dy_1 \ldots dy_m = \frac{(2\lambda)^m}{m!} \tag{A.6}$$

Proof: We prove this by induction on m. Let $I_m(\lambda)$ be the integral on the left-hand side. For $m = 1$, (A.6) is true because

$$I_1(\lambda) = \int_{|y_1| \leq \lambda} dy_1 = \int_{-\lambda}^{\lambda} dy_1 = 2\lambda \tag{A.7}$$

Assume that (A.6) is true for $m = j$. Consider $I_{j+1}(\lambda)$.

$$\begin{aligned}
I_{j+1}(\lambda) &= \int_{|y_1| + \ldots + |y_{j+1}| \le \lambda} dy_1 \ldots dy_{j+1} \\
&= \int_{-\lambda}^{\lambda} \left(\int_{|y_1| + \ldots + |y_j| \le \lambda - |y_{j+1}|} dy_1 \ldots dy_j \right) dy_{j+1} \\
&= \int_{-\lambda}^{\lambda} I_j(\lambda - |y_{j+1}|)\, dy_{j+1} \\
&= \int_{-\lambda}^{\lambda} \frac{[2(\lambda - |y_{j+1}|)]^j}{j!}\, dy_{j+1} \\
&= \int_{-\lambda}^{0} \frac{[2(\lambda + y_{j+1})]^j}{j!}\, dy_{j+1} + \int_{0}^{\lambda} \frac{[2(\lambda - y_{j+1})]^j}{j!}\, dy_{j+1} \\
&= 2\int_{0}^{\lambda} \frac{[2(\lambda - y_{j+1})]^j}{j!}\, dy_{j+1} \\
&= \frac{-2^{j+1}}{(j+1)!}(\lambda - y_{j+1})^{j+1} \Big|_{y_{j+1}=0}^{y_{j+1}=\lambda} \\
&= \frac{(2\lambda)^{j+1}}{(j+1)!} \qquad \text{(A.8)}
\end{aligned}$$

Therefore, (A.6) holds true for $m = j+1$ as well. By induction, it holds for any positive integer m.

Lemma A.3: For any vector γ, and any m m-dimensional vectors $L_1, \ldots, L_m$ such that L_k starts with $k-1$ zeros (which are followed by L_{kk}), the volume of the region defined by $S = \{\gamma + \sum \delta_k L_k : \sum |\delta_k| \le 1\}$ is $\frac{2^m}{m!}\left|\prod L_{kk}\right|$.

Proof: The volume of S is the absolute value of the integral

$$\int_S dz_1 \dots dz_m \tag{A.9}$$

where $z = \gamma + \sum \delta_k L_k$ because z is a point in S. This integral therefore equals

$$\int_{|\delta_1| + \dots + |\delta_m| \le 1} \frac{\partial(z_1, \dots, z_m)}{\partial(\delta_1, \dots, \delta_m)} d\delta_1 \dots d\delta_m \tag{A.10}$$

where

$$\frac{\partial(z_1, \dots, z_m)}{\partial(\delta_1, \dots, \delta_m)} = \begin{vmatrix} \frac{\partial z_1}{\partial \delta_1} & \dots & \frac{\partial z_1}{\partial \delta_m} \\ \dots & \dots & \dots \\ \frac{\partial z_m}{\partial \delta_1} & \dots & \frac{\partial z_m}{\partial \delta_m} \end{vmatrix} \tag{A.11}$$

is called the Jacobian of $z_1, \dots, z_m$ with respect to $\delta_1, \dots, \delta_m$ [Taylor 1992]. Because $z_j = \gamma_j + \sum \delta_k L_{kj}$, the partial derivative of z_j with respect to δ_k is

$$\frac{\partial z_j}{\partial \delta_k} = L_{kj} \tag{A.12}$$

Substituting this back into (A.11), we can see that the Jacobian is equal to $\det(L)$ if we treat L_k as the k-th column of a matrix L. Thus, (A.10) becomes

$$|L| \int_{|\delta_1| + \dots + |\delta_m| \le 1} d\delta_1 \dots d\delta_m \tag{A.13}$$

By Lemma A.2, the integral evaluates to $2^m/m!$ (simply let λ be 1). Moreover, L is lower-triangular because of the special form of its columns, L_k, as described in this lemma. As a result, the determinant of L is the product of its diagonal elements. Combining these two facts, we conclude that the integral (A.9) is $\frac{2^m}{m!}\prod L_{kk}$, and therefore the volume of S is the absolute value of (A.9): $\frac{2^m}{m!}\left|\prod L_{kk}\right|$.

Lemma A.4: For any vectors λ and μ such that $\lambda \le \mu$, and a lower-triangular matrix Φ with a unit diagonal, the volume of the region $\{y \,:\, \lambda \le \Phi y \le \mu\}$ is $\prod(\mu_k - \lambda_k)$.

Proof: The volume of the specified region is the absolute value of the integral

$$\int_{\lambda \le \Phi y \le \mu} dy_1 \ldots dy_m \qquad \text{(A.14)}$$

Notice that Φ is lower-triangular. Therefore, its determinant is the product of its diagonal elements, all of which are 1. Thus, $|\Phi| = 1$, and Φ is hence nonsingular. By a change of integration variables $z = \Phi y$, we see that the integral is equal to

$$\int_{\lambda \le z \le \mu} \frac{\partial(y_1, \ldots, y_m)}{\partial(z_1, \ldots, z_m)}\, dz_1 \ldots dz_m \qquad \text{(A.15)}$$

where $\frac{\partial(y_1, \ldots, y_m)}{\partial(z_1, \ldots, z_m)}$ is the Jacobian of $y_1, \ldots, y_m$ with respect to $z_1, \ldots, z_m$, as discussed in (A.11) [Taylor 1992]. Because $y = \Phi^{-1}z$,

$$\frac{\partial(y_1, \ldots, y_m)}{\partial(z_1, \ldots, z_m)} = \det\left(\Phi^{-1}\right) = \frac{1}{\det(\Phi)} = 1 \qquad \text{(A.16)}$$

The integral (A.15) is thus further reduced to

$$\int_{\lambda \le z \le \mu} dz_1 \ldots dz_m \qquad \text{(A.17)}$$

which is equal to $\prod(\mu_k - \lambda_k)$. The volume is the absolute value of the latter, but since $\lambda \le \mu$, all the factors $\mu_k - \lambda_k$ are non-negative. Consequently, the volume is simply $\prod(\mu_k - \lambda_k)$.

A.2 Volumes of Polyhedra

Equipped with the previous lemmas, we are now ready to prove the volumes of the relevant polyhedra and thus the ratio of allocated memory to memory actually used for array elements. First, consider the enclosing parallelepiped. It is defined by (4.7) on page 75, which is repeated below for convenience:

$$r_l \le Ry \le r_u \tag{A.18}$$

R is a lower-triangular matrix with a unit diagonal. By a direct application of Lemma A.4, the enclosing parallelepiped's volume is

$$V_E = \prod (r_{uk} - r_{lk}) \tag{A.19}$$

Next, let us construct a polyhedron within the image polyhedron as follows and compute its volume. For each k from 1 to m, let P_k be an m-dimensional vector within the transformed bounds such that

$$\begin{array}{ll} P_{k1} = c_1 \quad \dots \quad & P_{k,k-1} = c_{k-1} \\ P_{kk} = l_k(c_1, \dots, c_{k-1}) & \end{array} \tag{A.20}$$

where c is the center of symmetry of the image polyhedron, as computed in (4.24) on page 88 and $l_k(\dots)$ is the augmented bounds for the k-th dimension, as defined by (4.23) on page 87. Such a vector P_k must exist according to Lemma 4.1 on page 87 because the first k components of P_k satisfy the augmented bounds for the first k dimensions. (The choice of P_k is in fact the same as the z in the proof of Lemma 4.2.)

We claim that the convex hull spanned by all the P_k's and their mirror images about c is inside the image polyhedron. This follows from two facts. First, P_k's mirror image, like P_k itself, is in the image polyhedron because the image polyhedron is symmetric. Second, the convex hull they span is also in the image polyhedron because the image polyhedron is convex. We now find the volume of the convex hull, denoted V_H. By Lemma A.1, this convex hull can be written as

$$\{c + \sum \delta_k (P_k - c) \ : \ \sum |\delta_k| \le 1\} \qquad \text{(A.21)}$$

Because we choose P_k according to (A.20), the vector $P_k - c$ starts with $k-1$ zeros followed by $P_{kk} - c_k = l_k(c_1, \ldots, c_{k-1}) - c_k$. By Lemma A.3, the convex hull's volume is

$$V_H = \frac{2^m}{m!} \left| \prod_{k=1}^{m} [l_k(c_1, \ldots, c_{k-1}) - c_k] \right| \qquad \text{(A.22)}$$

Applying Lemma 4.3 on page 91 on each factor in the product gives us

$$V_H = \frac{1}{m!} \prod_{k=1}^{m} (r_{uk} - r_{lk}) \qquad \text{(A.23)}$$

Finally, having computed V_H and V_E, we can now compare the volumes of the image polyhedron and its enclosing parallelepiped. Let V_I be the volume of the image polyhedron. Since the convex hull defined above is inside the image polyhedron, $V_H \le V_I$. Moreover, from (A.19) and (A.23), we know that $V_E = (m!)\, V_H$. Hence,

$$V_E \le (m!)\, V_I \qquad \text{(A.24)}$$

In other words, the volume of the enclosing parallelepiped is at most $m!$ times that of the image polyhedron. Roughly speaking, the memory for the restructured array is at most $m!$ times that needed for data.

Appendix B

Additional Experimental Results

In this appendix, we briefly report experiments on other hardware platforms to demonstrate the generality of our previous observations. We used an IBM RS/6000 uniprocessor workstation and a Kendall Square Research KSR-2 shared address space multiprocessor. We ran essentially the same experiments as before, though with slightly different problem sizes in some cases because of differences in cache and TLB sizes.

B.1 Uniprocessor Workstation: IBM RS/6000

This section reports experimental results for an IBM RS/6000 Model 41T workstation based on the PowerPC 601 processor [Moore 1993; Paap and Silha 1993]. The PowerPC 601 has an on-chip, 32 KB, eight-way set-associative, write-back, unified cache. Cache lines are 64 bytes long but split into two 32-byte sectors for cache coherence purposes. In addition, the processor has a 256-entry, two-way set-associative TLB for instructions and data [Moore 1993]. The workstation itself also has a 512 KB second-level cache.

Figure B.1 (c.f. Figure 7.1 on page 138) shows the results for loops for which the prototype compiler considered array restructuring profitable; Figure B.2 (c.f. Figure 7.2 on page 139) shows those to which array restructuring was applied despite the compiler's decision.

Generally, these results are qualitatively the same as those for a DEC 3000 Model 400 workstation based on the Alpha 21064 processor, which have already been reported in

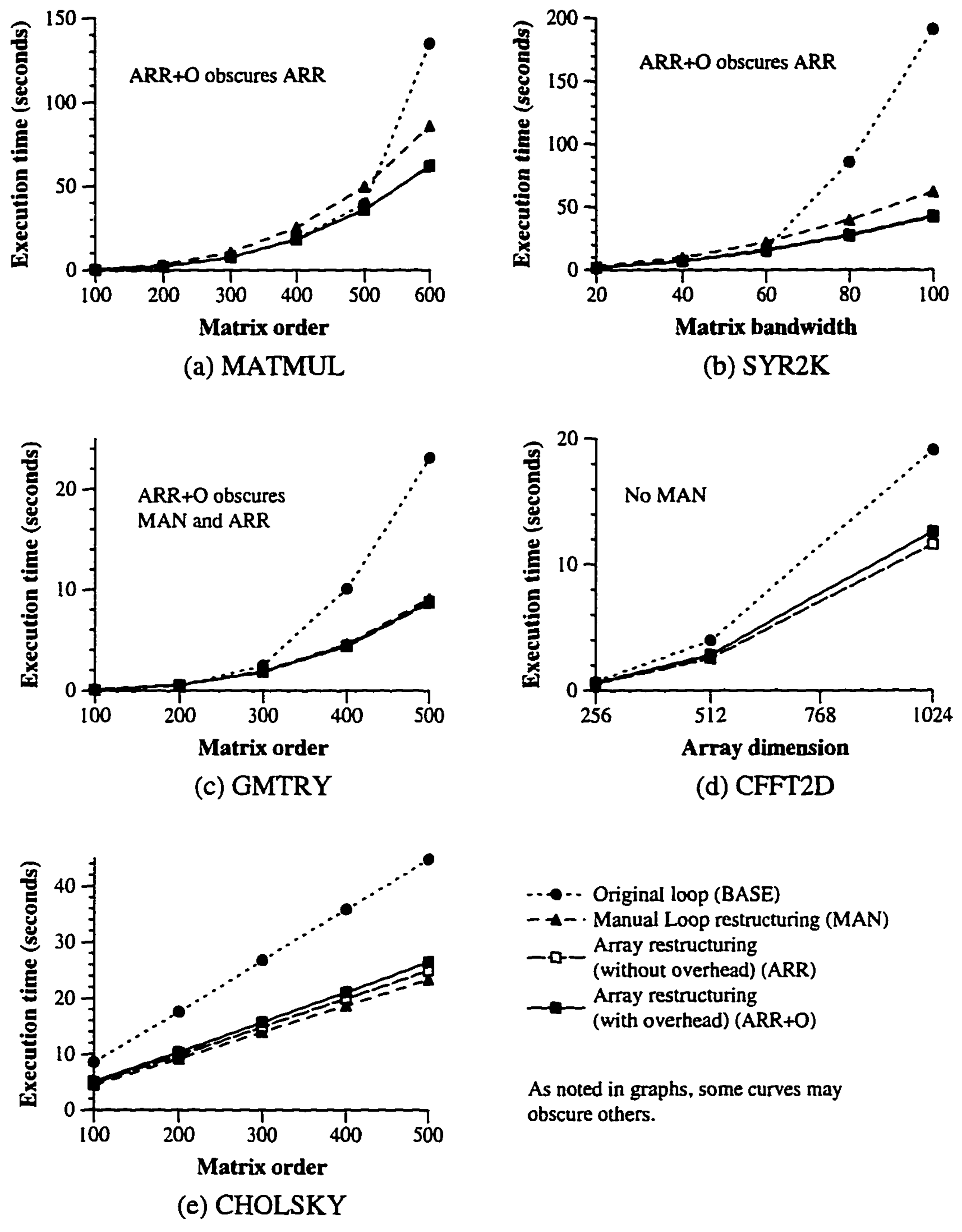


Figure B.1: Array Restructuring Performance on RS/6000 — Profitable Cases

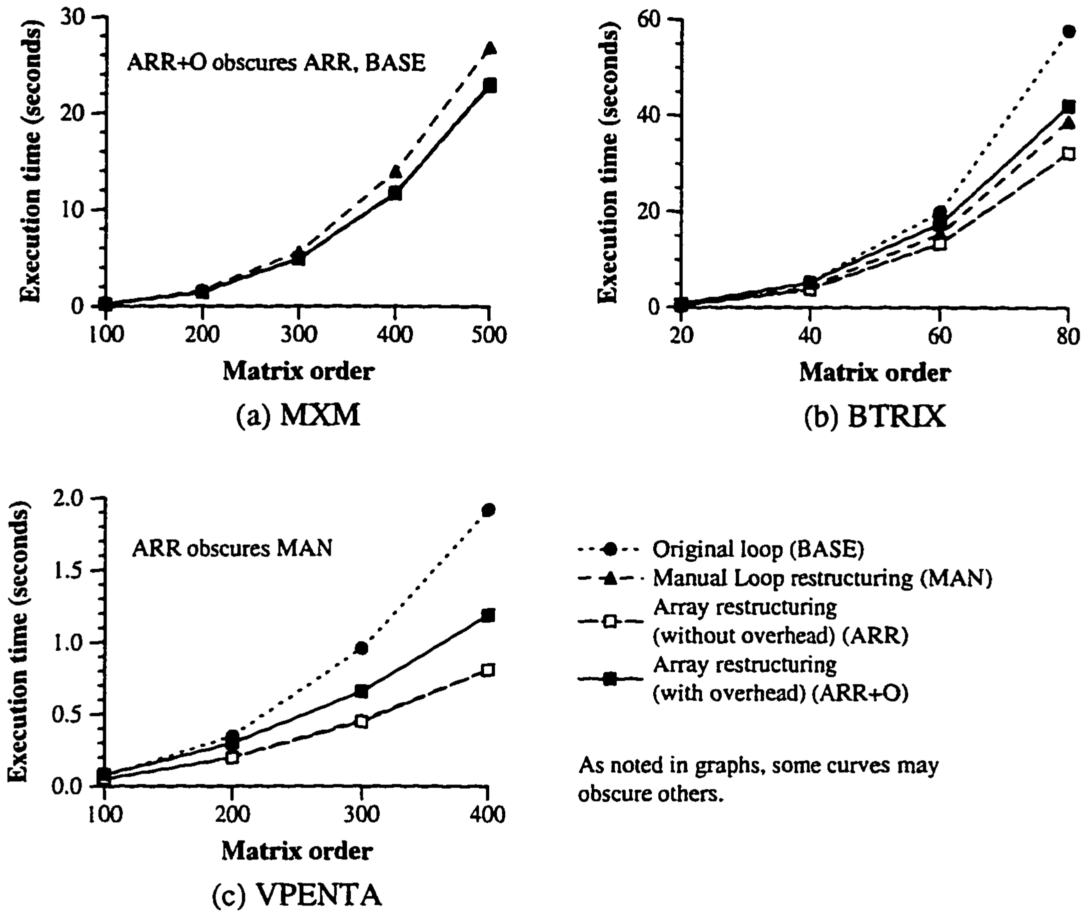


Figure B.2: Array Restructuring Performance on RS/6000 — Unprofitable Cases (with Manual Override of Compiler Decision)

Section 7.2. However, unlike in the DEC 3000 results, the execution times of MATMUL, SYS2K, and GMTRY rise markedly faster when the respective problem sizes exceed certain thresholds. Below these thresholds, array restructuring did not noticeably improve performance over the original loop, while loop restructuring sometimes even made performance slightly worse. Above the thresholds, both loop and array restructuring produced the expected improvement. We attribute this to the PowerPC 601's much larger TLB, which, with 256 entries, is eight times as large as the 32-entry TLB in an Alpha 21064 [DEC 1992]. A larger TLB, or cache, implies a larger problem size threshold above which locality improving optimizations matter to performance — and below which anticipated performance gains from such optimizations may not materialize. For example, from Figure B.1(b), SYR2K's problem size threshold appears to be 60. This is almost exactly eight times — the ratio of TLB sizes — the threshold for the Alpha processor (namely, 7 or 8), which has been predicted hypothetically in Section 7.2.2 as well as observed experimentally in Figure 7.5 on page 147.

B.2 Shared Address Space Multiprocessor: KSR-2

In this section, we report experiments on a Kendall Square Research KSR-2, a shared address space multiprocessor. KSR-2 has a cache-only memory architecture (COMA): it has no "main memory;" all its memory is distributed to processors and kept coherent by hardware [KSR 1992]. Processors are connected in a hierarchy of rings, each with at most 32 processors. Each processor has a 32 MB, 16-way set-associative *local cache*. Local cache space is allocated in *pages* of 16 KB, but different local caches are kept coherent in *subpages* of 128 bytes. Thus, the cache line size is 16 KB for allocation purposes, but 128 bytes for coherence purposes. Each processor also has a 512 KB, 2-way set-associative *subcache*, divided into two halves for instructions and data. Like the local cache, the subcache has different units for allocation and for coherence: 2 KB *blocks* and 64 bytes *subblocks* respectively [KSR 1992]. Misses in the subcache and local cache lead to latencies of 20-24 cycles (0.5-0.6 μs) and 175-600 cycles (4.375-15 μs) respectively [KSR 1993]. Our experiments were done on a KSR-2 with two rings. To avoid potential complications if data traffic crosses the ring boundary, we used only the 25 processors on one ring[1].

Results are shown in Figure B.3 and Figure B.4 respectively for cases where array restructuring was applied according to and in spite of compiler decision. They resemble results for the SGI Power Challenge, shown in Figure 8.1 on page 158 and Figure 8.2 on

1. Each ring was configured for 32 processors, but only 25 processors were operational at the time of our experiments.

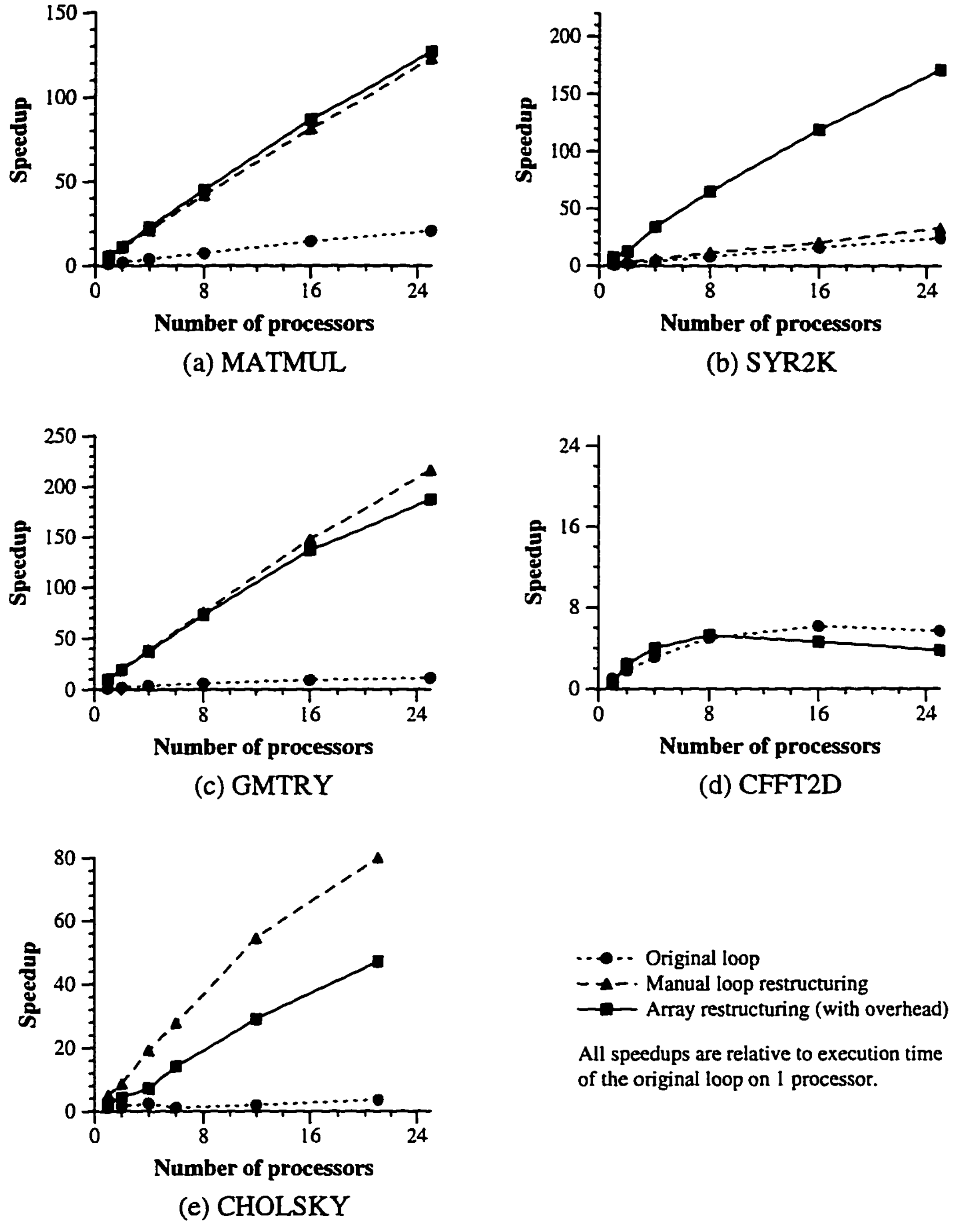


Figure B.3: Parallel Speedups on KSR-2 — Profitable Cases

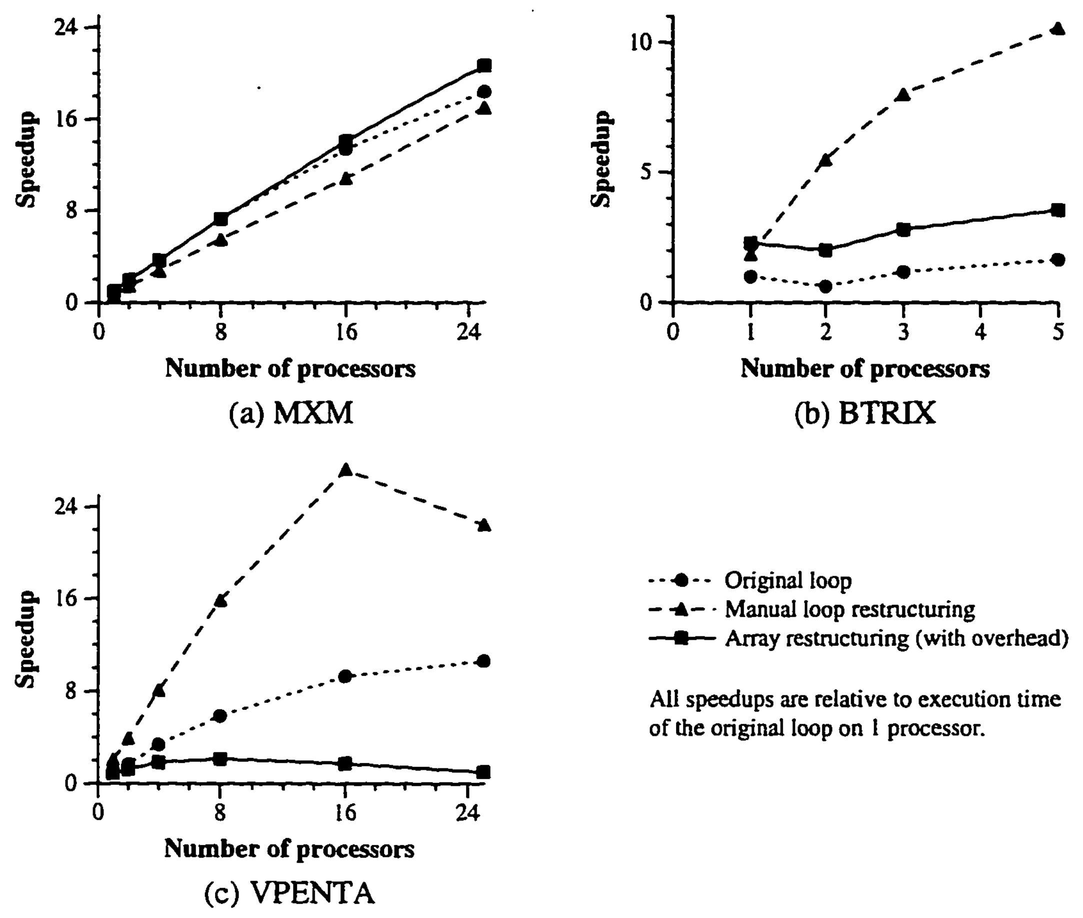


Figure B.4: Parallel Speedups on KSR-2 — Unprofitable Cases
(with Manual Override of Compiler Decision)

page 159. All speedups are relative to the execution time of the original loop running on one processor, for reasons explained earlier in Section 8.1.

Qualitatively, results for the two machines are similar. Quantitatively, however, both loop and array restructuring improved performance much more significantly on the KSR-2 than on the Power Challenge. Notice that the vertical scales of the KSR-2 graphs are often at least several times those of the corresponding Power Challenge graphs so as to accommodate the large speedups relative to the original loop that we observed on the KSR-2. We attribute this to KSR-2's much larger cache line sizes. Since cache space is allocated in very large units (2 KB subcache blocks, 16 KB local cache pages), each cache contains

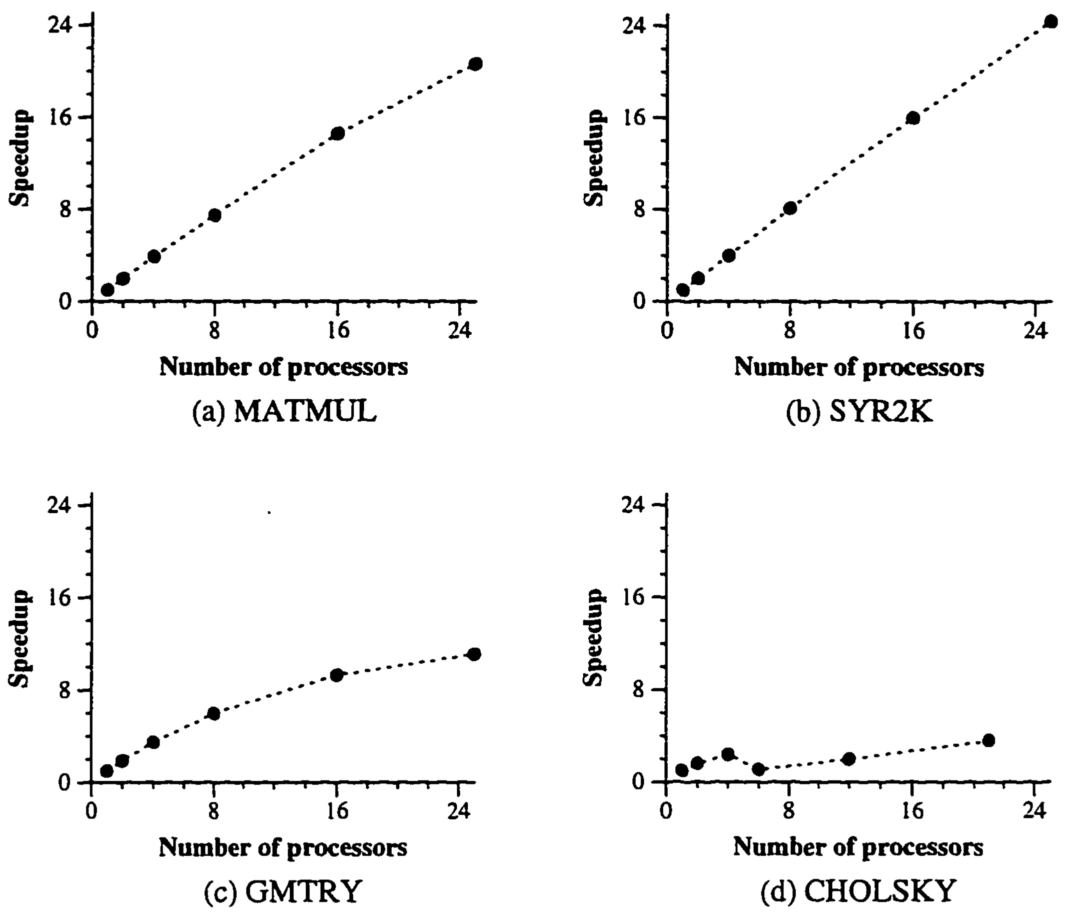


Figure B.5: Parallel Speedups of Original Loop on KSR-2 — Selected Cases

relatively few such units. This in turn leads to frequent cache misses when memory is accessed at long strides, as in the original loops. Moreover, replacing a cache line is especially costly because it potentially displaces a large amount of data. (There is no need to load the same large amount of data, though, because coherence is managed in smaller units of subblocks and subpages.)

Figure B.5 shows the speedups of the original loops of MATMUL, SYR2K, GMTRY, and CHOLSKY. These curves are the same curves in Figure B.3, only shown on different vertical scales because those in Figure B.3 make the rise in speedup illegible. With these graphs, we again observe the same trend as we do for the Power Challenge in Section 8.2.

For MATMUL and SYR2K, the original loop parallelized reasonably well. Array restructuring improved performance by roughly the same factor on any number of processors because it improved the locality of execution on each individual processor independently. For GMTRY and especially CHOLSKY, however, the original loop parallelized poorly in part because of false sharing. Both loop and array restructuring remedied the problem; both produced performance that was not only higher in absolute terms but also scaled better with the number of processors.

Vita

Shun-Tak Albert Leung received the Bachelor of Science in Engineering degree, with First Class Honors, in 1986 and the Master of Philosophy degree in 1990, both in Electrical Engineering from the University of Hong Kong. He received the Master of Science degree in 1992 and the Doctor of Philosophy degree in 1996, both in Computer Science and Engineering from the University of Washington.