CDA: Computation Decomposition and Alignment

by

Dattatraya H. Kulkarni

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy.
Graduate Department of Computer Science.
in the University of Toronto

© Copyright by Dattatraya H. Kulkarni 1997

National Library of Canada

Acquisitions and Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada

Acquisitions et services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-27983-9

Canada

CDA: Computation Decomposition and Alignment

Dattatraya H. Kulkarni

Doctor of Philosophy, 1997

Department of Computer Science

University of Toronto

Abstract

Restructuring compilers have been effective in tailoring nested loops and arrays so as to improve performance on both uniprocessor and multiprocessor systems. The regular structure of nested loops and arrays has enabled their systematic analysis and transformation. The focus of this dissertation is a new and generalized loop transformation framework, called *Computation Decomposition* and *Alignment* (CDA).

The *linear loop transformation* framework introduced in 1990 was a major breakthrough, partly because it provided a unified view of many of the earlier loop transformations, and partly because it was a formal method based on linear algebra. Since then, the compiler community has designed algorithms which automatically derive linear transformations that achieve specific optimization objectives in given nested loops. The framework also sparked the development of generic techniques to derive efficient code for the linearly transformed loop structures.

The main contribution of this dissertation is the CDA transformation framework, which is capable of restructuring nested loops at the granularity of statements and subexpressions. The granularity of transformation is thus finer than in the linear loop transformation framework, which transforms nested loops at the granularity of entire iterations. A CDA transformation is applied to a nested loop in two steps. First, the loop iteration space is decomposed into multiple *computation spaces*, each representing computations of a statement or a subexpression. Second, each of the computation spaces is linearly transformed with a (possibly) different transformation matrix.

CDA unifies into a single framework many existing transformations including all linear loop transformations. A linear loop transformation only modifies the execution order of the iterations, while a CDA transformation modifies both the *composition* of the iterations and the execution order of the re-composed iterations. This feature enables new optimizations which cannot be obtained by linear loop transformations alone. In this thesis, we show how CDA transformations can achieve the effect of certain global data transformations. We present heuristic algorithms to automatically derive CDA transformations to reduce *i*) the number of cache conflicts and *ii*) the

number of ownership tests, and we show how CDA can achieve several other optimizations. We also compare the performance of some benchmark loops to the corresponding CDA transformed loops using a simulator and three different types of real computer systems.

Acknowledgments

First. I would like to thank Michael Stumm for instilling in me qualities essential in an independent researcher. I will no doubt spend years learning his ability to distill myriad of information into a crisp paragraph.

As a supervisor during my first year in the department. Hector Levesque encouraged me to freely pursue any area that interested me. Ken Sevcik. Tarek Abdelrahman and Christina Christara provided valuable feedback that greatly improved the quality of the thesis. Discussions with them in meetings and corridors have contributed in some or the other way to the direction my work has taken. I thank Vivek Sarkar. my external examiner. for careful reading and suggestions that surely will have a positive influence on my future research plans. I thank Charles Clarke for careful reading of the thesis even within relatively short time.

I thank Keeranur Kumar at IBM T.J. Watson for introducing me to the exciting world of compilers and for being a mentor. friend and family bundled in one. I am indebted to my teachers during all stages of my schooling for recognizing my strengths and ironing away my weaknesses.

Many friends and colleagues have brightened my life in Toronto. Sudarsan. Shankar and Ravindran have always been there for me. Brian. Daniel. Hui. Jaseemuddin. Jeannine. Karim. Luis. Shailesh. Steve. and Yiming have often helped regain my sanity.

Radha-Sudarsan. Jeannine-Alan. Lalita-Kuppu. Prabha-Pawan. Sharada-Parameshwar. Rati-Prakash. Sujata-John and Nalawadis' have provided me family thousands of miles away from home. Thank you.

I have been lucky to have a large number of friends in India and North America who have congratulated me on successes and consoled me on failures. I cannot imagine living through the graduate school frustrations without the reassuring calls from Anand. Dattaraj. Nagaraj. Narayan. Naren. Sharad. and Vinayak.

My parents and family have relentlessly cheered me on on every single day of nearly a quarter century of schooling. My dad. my first Guru. has always been a source of strength and insight into life which have made me the person I am today. My brother has been the master planner of my academic career. He has always believed in my ability even when I had doubts during graduate school. I thank my sisters for not asking the questions a graduate student dreads most — "When are you going to graduate?". Finally. Suhasini brought the first ray of light when the end was in sight and has cheered me on through the crucial last miles of the tunnel that had otherwise seemed endless.

To my loving parents and my brother

Contents

1 Introduction 1

1.1 Linear Loop Transformations ... 3

1.2 Computation Decomposition and Alignment Transformations ... 5

1.3 Contributions of the Dissertation ... 7

2 Linear Loop Transformation Framework 9

2.1 Representation of the Loop Structure ... 9

2.1.1 Model of the Loop Structure ... 10

2.1.2 Representation of Array References ... 11

2.1.3 Representation of Loop Bounds ... 12

2.2 Data Dependence Analysis ... 14

2.3 Mathematics of Linear Loop Transformations ... 16

2.3.1 Basic Transformation Technique ... 17

2.3.2 Derivation of New Loop Bounds ... 18

2.4 Advantages of Linear Loop Transformations ... 21

2.5 Techniques to Derive Linear Loop Transformations ... 24

2.5.1 Deriving Canonical Loops ... 24

2.5.2 Dependence Internalization ... 25

2.5.3 Access Normalization ... 26

2.5.4 Balancing Processor Load ... 28

3 Computation Decomposition and Alignment Framework 30

3.1 Overview of CDA Transformation Framework ... 30

3.2 Representation of the Loop Structure ... 32

3.3 Computation Decomposition 33

3.4 Computation Alignment 37

3.5 Generating New Loop Bounds 41

3.6 Applications of CDA 46

3.7 Disadvantages of CDA 47

4 Optimizing CDA Transformed Loops 48

4.1 Removing Empty Iterations 50

4.2 Reducing the Overhead of Guard Computations 54

4.2.1 Incremental Removal of Guards 55

4.2.2 Partitioning the Iteration Space into Homogeneous Segments 61

4.2.3 Optimizing the Evaluation of Guards 62

4.3 Optimization of Space Overhead for Temporaries 63

5 Application of CDA to Reduce Number of Cache Conflicts 65

5.1 Reducing the Number of Cache Conflict Misses 65

5.2 Representation of Cache Conflicts 67

5.3 Derivation of a Suitable CDA Transformation 68

5.3.1 Initial Computation Decomposition 69

5.3.2 Deriving the Computation Alignment 70

5.4 Comparison of CDA with Padding 76

6 Application of CDA to Remove Ownership Tests from SPMD Codes 79

6.1 *P-computes*: Flexible Computation Rules 80

6.2 Removing Ownership Tests in P-computes rules 82

6.3 Derivation of CDA Transformation to Pack Iterations 85

6.3.1 Derivation of Computation Decomposition 85

6.3.2 Derivation of Computation Alignment 86

6.3.3 Data Alignment of Temporary Arrays 90

6.3.4 Summary of Stages to Pack Iterations 90

6.4 Scanning Local Iteration Space 91

7 Other Applications of CDA 94

7.1 Improving Instruction Level Parallelism 94

7.2 Eliminating Synchronizations 96
7.3 Generalizing Loop Distribution 99
7.4 Transforming Imperfect Loop Nests 103
7.5 Using CDA to Improve Global Optimization 105
7.6 Summary 106

8 Application of CDA to Example Nested Loops **107**
8.1 Reducing the Number of Cache Conflicts 108
8.1.1 *Rtmg* Loop 109
8.1.2 *Mg* Loop 112
8.1.3 *Vpenta* Loop 117
8.2 Removing Ownership Tests 120
8.2.1 *Wanal* Loop 121
8.2.2 *Swm* Loop 123

9 Concluding Remarks **126**
9.1 Summary 126
9.2 Future Work 128
9.3 Epilogue 130

A A Catalog of Loop Transformations **132**
A.1 Preliminary Transformations 132
A.2 Primary Transformations 135
A.3 Secondary Transformations 140

List of Figures

1.1 A loop transformation to expose parallelism. 2

1.2 A loop transformation to improve memory access behavior......................... 2

1.3 The linear loop transformation framework. 4

1.4 The CDA transformation framework. 5

1.5 A CDA transformation of a 2-dimensional loop.................................... 6

2.1 Model for perfectly nested affine loops.. 10

2.2 Nested loop L.. 11

2.3 Dependences in iteration space... 16

2.4 The transformed iteration space of loop L. 21

2.5 The transformed loop L. 21

2.6 Example transformation matrices for two dimensional loops. including (a) reversal of outer loop. (b) reversal of inner loop. (c) reversal of both loops. (d) interchange. (e.f) skew by p in second and first dimensions. and (g) wavefront respectively. 22

2.7 Dependence internalization in two dimensional loop. 25

2.8 An example access normalization... 27

2.9 Choice of iterator to partition for good load balance............................ 28

3.1 Mapping of portions of iterations in CDA. For clarity. this figure shows how the computations of a single original iteration are mapped onto new iterations. 31

3.2 An example CDA transformation.. 31

3.3 The program model 33

3.4 Running example and new loop body after Computation Decomposition........... 35

3.5 Computation spaces for the running example. 37

3.6 Illustration of a simple Computation Alignment of the computation spaces. 39
3.7 Algorithm CDA-bounds to derive new loop bounds. 42
3.8 CDA transformed running example loop. 43
3.9 An example loop used to illustrate overhead of empty iterations. 44
3.10 Deriving new loop bounds. 45
3.11 Transformed loop after simple guard optimizations. 46

4.1 The loop used to illustrate the effect of techniques to reduce overheads. 49
4.2 Overheads in a CDA transformed loop, called *Loop 1*, with offset alignment (k, k). In the bar chart above, the bars on the left correspond to the execution times of *Loop 1* with overheads, whereas the bars on the right correspond to the execution times of *Loop 1* after reducing overheads with techniques described in this chapter. . 49
4.3 Overheads in a CDA transformed loop with a linear alignment, called *Loop 2*. 50
4.4 Shapes of iteration spaces 51
4.5 Tight transformed bounds 52
4.6 Empty iterations in an iteration space with tight bounds. 53
4.7 Performance benefits of eliminating empty iterations. 54
4.8 Transformed computations spaces to illustrate steps in algorithm *CDA-guard-rem*. The transformed loop corresponding to the transformed computation spaces is called *Loop 3*. 56
4.9 Merging Guards for statements $S_1, \ldots, S_K$. 57
4.10 Remove guard computations in L. 58
4.11 Generation of code for subnests. 60
4.12 Performance benefits of removing guards by *CDA-guard-rem*. 61
4.13 Partitioning union into homogeneous segments. 61
4.14 Performance benefits of scanning in homogeneous segments. 62

5.1 Reducing cache conflicts with modification to array layout and CDA. 67
5.2 Conflict graph for statement S. 68
5.3 Algorithm *A1* derive initial Computation Decomposition. 70
5.4 Initial decompositions of conflict graph for statement S. 71
5.5 Algorithm *A2* to derive alignments that reduce cache conflicts. 73

6.1 SPMD codes for an example loop. 83
6.2 Algorithm *B1* to decompose statements. 86
6.3 Computation Decomposition of a loop with $\odot$ operators. 87
6.4 Algorithm *B2* to derive Computation Alignment. 88
6.5 Computation Alignment of a loop with $\odot$ operators. 90
6.6 Algorithm *B3* to data align temporary arrays. 91
6.7 Modification of loop bounds to scan local iteration space. 93

7.1 Application of CDA transformation to improve instruction level parallelism. 95
7.2 CDA transformation to eliminate barrier synchronization. 98
7.3 An example loop distribution. 99
7.4 A loop that cannot be distributed. 100
7.5 Loop distribution as a CDA transformation. 100
7.6 An example of breaking dependence cycles to enable loop distribution. 101
7.7 CDA transformation for partial loop distribution. 102
7.8 CDA transformation for partial loop distribution. 103
7.9 Converting a simple imperfectly nested loop into a perfectly nested loop. 104
7.10 CDA transformation of an imperfectly nested loop. 104
7.11 Effect of CDA on global optimization. 105

8.1 Conflicting references in the original rtmg loop. 109
8.2 The number of cache misses in the original and the CDA transformed *rtmg* loops with array sizes of 64x64. 110
8.3 Execution time of *rtmg* loop on a SPARC 10 workstation. 111
8.4 Execution time of *rtmg* loop on a SPARC 10 workstation for varying data sizes. 111
8.5 Cache misses in the original and the CDA transformed *mg* loop. 113
8.6 Conflict graph for the original *mg* loop. 113
8.7 Conflict graph for the CDA transformed *mg* loop. 114
8.8 Number of additional array elements in the CDA transformed and the array padded *mg* loops. 116
8.9 Execution time of *mg* loop with 256x256x256 arrays on a SPARC 10 workstation and a single KSR1 processor. 117
8.10 Cache misses in the original and the CDA transformed *vpenta* loop. 120

8.11 Execution time of *vpenta* loop on SPARC 10 workstation, a processor of the KSR1 and RS/6000 workstation. 120
8.12 Number of additional array elements required for the CDA transformed and the array padded *vpenta* loops. 121
8.13 Execution time of *wanal* loop on KSR1. 122
8.14 Execution time of *swm* loop on KSR1. 125

Chapter 1

Introduction

> There is nothing permanent, except change.
> — *Heraclitus*

Restructuring compilers modify program structure in order to improve performance on target hardware. On uniprocessors, restructuring compilers are often used to maximize cache hit rates so as to hide latencies of the memory hierarchy. On parallel computer systems, restructuring compilers are often used to parallelize the code and to maximize data access locality. Parallelizing code includes the identification of the parallelism in the code, the even distribution of computations onto processors, and the placement of data to match the mapping of parallel computations (or vice versa). Improving data access locality includes maximizing cache hit rates and minimizing the number of remote accesses.

The restructuring techniques that have been proposed over the years focus primarily on nested loops and arrays — loops, because they are a regular, well defined control structure that are straightforward to manipulate and because they constitute the core of many applications — arrays, also because they have a regular structure. The structure of a loop and the layout of data in memory greatly affect performance:

- The loop structure determines the data flow relations between loop iterations, and hence the amount of parallelism available in the loop.
- The loop structure and data layout determine the patterns in which memory is accessed, and hence the spatial and temporal cache locality as well as the number of remote memory accesses required.
- The loop structure affects the number of iterations mapped to each processor, and hence the balance of computational load.

For this purpose, restructuring compilers (among other things) apply loop and data transformations to modify the structure of nested loops and change the data layout so that the same result

```
for i = 1,n                                  for i = 2,2n
  for j = 1,n                                  for j = max(1,i-n),min(n,i)
    S: A(i,j) = A(i-1,j) + A(i,j-1)              ij = i-j
  end for                        ==>             S: A(ij,j) = A(ij-1,j) + A(ij,j-1)
end for                                        end for
                                             end for
```

Figure 1.1: A loop transformation to expose parallelism.

```
for i = 1,n                                  for i = 0,n-1
  for j = 1,i                                  for j = 1,n-i
    S: A(i-j,j) = A(i-j,j) + B(i-j,j)            S: A(i,j) = A(i,j) + B(i,j)
  end for                        ==>           end for
end for                                      end for
```

Figure 1.2: A loop transformation to improve memory access behavior.

is computed but with improved performance.

Consider the nested loop on the left hand side of Figure 1.1. The loop, as is, cannot be directly mapped onto a multiprocessor to run in parallel. Every execution of statement S must wait for the results of the previous execution of S. A loop transformation can, however, re-arrange the execution order of the iterations so that each iteration requires data produced only by earlier i iterations, allowing the j iterations to be executed in parallel. The loop on the right hand side of Figure 1.1 is a transformation of the loop on the left hand side. It computes the same result, but exhibits improved parallelism.

Loop transformations can also be used to improve cache and memory access behavior. Consider the two dimensional loop on the left hand side of Figure 1.2. This loop has poor cache locality, because a new cache line is accessed in every iteration, and successive iterations do not use elements of previously accessed cache lines. The loop can be transformed into the loop on the right hand side of the figure, which is semantically equivalent to the original loop. The transformed loop performs better than the original loop on both uniprocessors and multiprocessors if the matrices are stored in row major order, because the elements of a cache line are used in successive iterations.

The example of Figure 1.2 also illustrates how a loop transformation can reduce the number of remote memory accesses in the case where the loop is run in parallel on a multiprocessor. Consider a parallel system with p processor-memory module pairs and a mapping scheme where the outer loop is executed in parallel such that all j iterations of an outer iteration i execute on processor

$i \ mod \ p$.[1] Also assume that arrays A and B are mapped onto memory modules such that row k of each array resides on memory module $k \ mod \ p$. With the original loop, processor k will need to access array elements $A(i-j, j)$ and $B(i-j, j)$ remotely whenever $(i-j) \ mod \ p \neq k$. With the transformed loop, however, all array accesses in the loop are local. Hence, the transformation increased locality, given the initial mapping of computation and data.[2]

This dissertation will focus on loop transformation techniques. The above examples show that the structure of a loop is critical for good performance.

1.1 Linear Loop Transformations

Over the years, many loop transformations have been developed and proposed — see Bacon et al. for a good overview [6]. Prior to 1990, it was found to be difficult for a compiler to determine how and in what order to combine and apply a sequence of transformations. Without a formal framework, reasoning about the effects of a sequence of transformations is ad hoc, making the design of algorithms to automatically transform loops difficult. Without a formal representation of the entire transformation sequence, code generation is difficult as well, often producing complex loop bounds. Hence, there was a need for a formal transformation framework that provides a mathematical basis for effectively:

i) *representing* a set of loop transformations in a concise and uniform way,

ii) *reasoning* about and comparing alternative loop transformations and their effects using formal methods,

iii) automatically *deriving* loop transformations that achieve specific optimization objectives, and

iv) *applying* loop transformations (i.e. generating the code for the transformed loop) in an automated way.

Fortunately, well structured loops are amenable to such a mathematical formalization. In this regard, the *linear loop transformation* framework, introduced in 1990 [10, 26, 29, 36, 54], was a major breakthrough that greatly simplified the task for the compiler, partly because it was a formal method based on linear algebra and partly because it provided a unified view of many of the previously proposed loop transformations. With this framework, it became possible to design algorithms that automatically search for transformations for given optimization objectives.

[1] There are no data flow constraints for parallel execution, since all iterations are independent.

[2] Note that a column major storage order will produce a completely different memory access behavior.

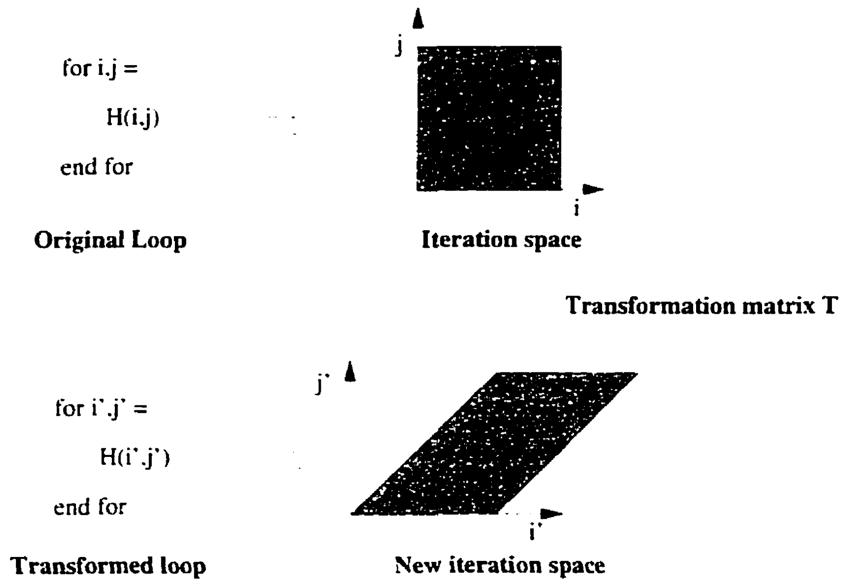


Figure 1.3: The linear loop transformation framework.

In the linear loop transformation framework, iterations of a nested loop are represented by an *iteration space*, which is an integer space bounded by hyperplanes corresponding to the loop bounds. A non-singular integer matrix is used to linearly transform the iteration space into a new iteration space.[3] The new iteration space then corresponds to a new (now transformed) loop. The relationship between the loops, iteration space and the transformation matrix is illustrated in Figure 1.3.

The introduction of the linear loop transformation framework was a significant contribution in that *i*) it allowed a single matrix to represent a compound transformation of many existing transformations such as skew, reversal, interchange etc. *ii*) it allowed the development of a set of generic techniques to generate transformed loop structures in a systematic way, independent of the particular sequence of transformations being applied, and *iii*) it sparked the development of algorithms to derive transformation matrices that achieved specific objectives for a given nested loop.

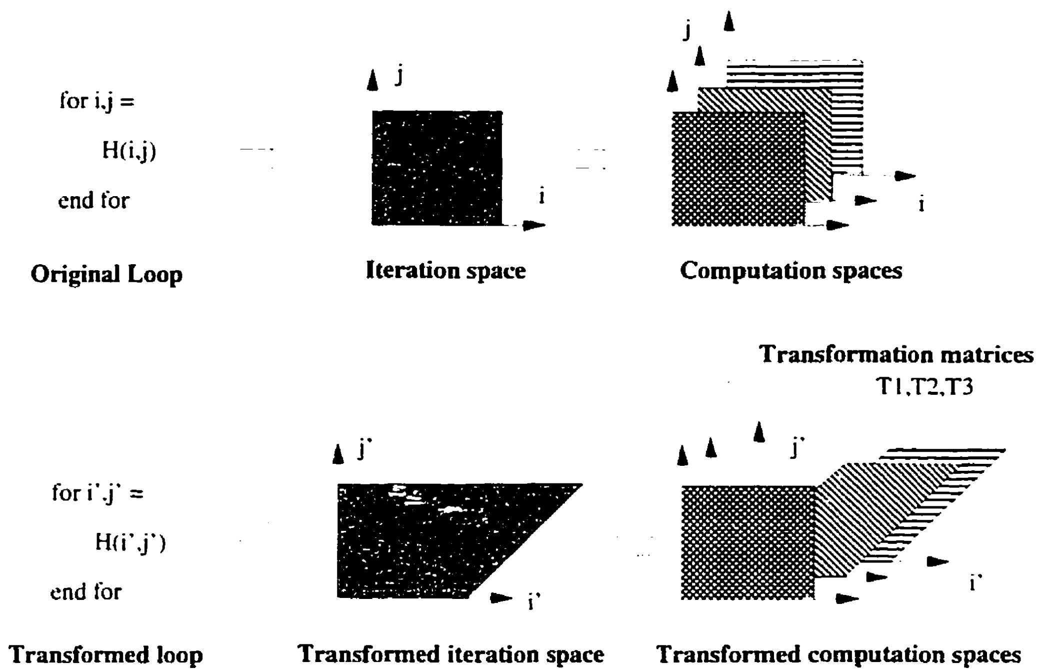


Figure 1.4: The CDA transformation framework.

1.2 Computation Decomposition and Alignment Transformations

The main contribution of this dissertation is an extension to the linear loop transformation framework called the Computation Decomposition and Alignment (CDA) transformation framework. The objective is to unify into a linear algebraic framework a larger number of loop optimizations than possible within the linear loop transformation framework. A CDA transformation maps the computations of an original nested loop into computations in a new, transformed loop. It is applied in two steps as shown in Figure 1.4:

- First, the iteration space of the loop is decomposed into multiple integer spaces, called *computation spaces*, each representing computations of a statement or a subexpression.
- Second, each of the integer spaces is linearly transformed with a (possibly) different transformation matrix.

The transformed computation spaces together define the transformed iteration space, and thus, the new CDA transformed loop. It should be noted that a linear loop transformation is a spe-

[3] The linear loop transformation framework was originally introduced in the form of the *unimodular* loop transformation framework, where the transformation matrices must have a determinant of ± 1. The linear loop transformation framework is a generalization of the unimodular loop transformation framework, where the transformation matrices may have any non-zero determinant (including ± 1).

```
for i = 1,n
   for j = 1,n
      A(i,j) = B(i,j) + A(i,j) + B(i+1,j)
   end for
end for
```

$\Longrightarrow$

```
for i = 0,n
   for j = 1,n
      if (i < n) then
         t(i+3,j) = B(i+1,j)
      if (i > 0) then
         A(i,j) = t(i+2,j) + A(i,j) + B(i+1,j)
   end for
end for
```

Figure 1.5: A CDA transformation of a 2-dimensional loop.

cial CDA transformation which does not first decompose the iteration space. However, the CDA transformation framework differs from the linear loop transformation framework in several ways:

(*i*) The linear loop transformation framework restructures loops at the granularity of iterations, whereas the CDA transformation framework can restructure loops at a *finer granularity* of statements and subexpressions.

(*ii*) A linear loop transformation is defined by a single transformation matrix, whereas a CDA transformation requires potentially *several transformation matrices*, one for each computation space.

(*iii*) A linear loop transformation only modifies the execution order of the iterations, while CDA transformations modify both the *composition* of the iterations and the execution order of the re-composed iterations.

(*iv*) The *search space* for legal CDA transformations of a nested loop is considerably larger than that for legal linear loop transformations.

(*v*) The CDA transformation framework *unifies* additional loop transformations besides those unified by the linear loop transformation framework.

As an example of a CDA transformation, Figure 1.5 shows a loop that is CDA transformed to the loop on the right hand side. The transformed loop has a loop structure that is substantially different than that of the original loop, and cannot be obtained by a direct application of any existing loop transformation technique. The CDA transformation applied here improves the cache utilization by reducing the number of cache conflicts that occur with certain certain cache and array sizes.

1.3 Contributions of the Dissertation

The main objective of this dissertation is to show that there exist opportunities to restructure loops at finer computation granularity than there exist within the linear loop transformation framework. The final outcome is a transformation framework that is more effective than the linear loop transformation framework and yet preserves its elegance. The main contributions of this dissertation are:

1. We develop the basic CDA transformation framework, capable of restructuring nested loops at the granularity of statements and subexpressions (Chapter 3).

2. An undesirable effect of CDA transformations is that the transformed loop has computational and spatial overhead which a pure linear loop transformation would not have. We describe techniques to optimize the CDA transformed loops by reducing these overheads (Chapter 4).

3. We identify several situations in which the performance of nested loops can be improved by restructuring at statement and subexpression granularity. But because increased flexibility for transformation within the CDA framework comes at a cost of vastly larger search spaces, heuristics that use the knowledge about the optimization context are key in deriving CDA transformations efficiently.

4. We present an algorithm capable of heuristically deriving CDA transformations that optimize nested loops in the context of reducing the number of cache conflicts (Chapter 5).

5. We present an algorithm capable of heuristically deriving CDA transformations that optimize nested loops in the context of improving the efficiency of SPMD code on multiprocessors (Chapter 6).

6. We illustrate the utility of CDA in improving the performance of nested loops in several other contexts, including the context of increasing instruction level parallelism and of reducing the number of barrier synchronizations required in parallel code (Chapter 7).

7. We illustrate the application of the CDA transformation techniques using example loop nests from benchmarks. We compare the performance of the original and the CDA transformed versions of the loops by executing them on both uniprocessor and multiprocessor platforms (Chapter 8).

We begin by describing the linear loop transformation framework in the next chapter. since the CDA framework is a direct extension of that.

CHAPTER 2

Linear Loop Transformation Framework

> What need have I of magic charms–
> 'Abracadabra!' and 'Prestopuff'? ...
> Transformed would I be to toad or lizard
> — *Robert Graves, Love and Black Magic*

The linear loop transformation framework [10, 26, 29, 36, 48, 54] provides a formal and unified basis for numerous loop transformations - loop *reversal*, *interchange*, *permutation*, *skew*, *scaling*. The framework has been effective and has led to the development of many algorithms and tools for optimizing compilers. In this chapter, we describe methods underlying the linear loop transformation framework because CDA is an extension of the framework to finer computation granularity.

We first describe the representation of nested loops and outline the representation of data flow between the iterations of nested loops. We show how a linear transformation modifies the loop structure and the data flow, and we present techniques to derive the transformed nested loop given a transformation. We also illustrate how the linear algebraic representation in the framework can be utilized to systematically and automatically derive transformations to achieve specific optimization objectives.

2.1 Representation of the Loop Structure

In this section we describe the model of nested loops that can be linearly transformed. We then present the linear algebraic representation of the loop bounds and the array references in the loop. This representation of nested loops forms the basis for the analysis and transformation techniques discussed in the subsequent sections.

```
for I_1 = L_1, U_1
   for I_2 = L_2(I_1), U_2(I_1)
      ...
         for I_n = L_n(I_1, ..., I_{n-1}), U_n(I_1, ..., I_{n-1})
            H(I_1, ..., I_n)
         end for
      ...
   end for
end for
```

Figure 2.1: Model for perfectly nested affine loops.

2.1.1 Model of the Loop Structure

The linear loop transformation framework assumes *perfectly nested affine loops*, so they have the structure depicted in Figure 2.1. This loop structure is sufficiently general to represent many common nested loops. We say that the loop is n-dimensional, since the nest has n loop statements. The body of the loop nest is denoted by H, which may be a sequence of statements including those that contain additional loops. $I_1, \ldots, I_n$ are the iterators. L_i and U_i are the lower and upper loop limits or loop bound expressions for iterator I_i; they are assumed to be linear functions of $I_1, \ldots, I_{i-1}$. The loop is assumed to be normalized, so all iterators have a stride of one.[1]

The notion of perfect nesting characterizes well structured loops for which analysis and transformation techniques are relatively simple. The loop nest formed by iterators I_1 to I_n is perfectly nested iff for all iterators I_k such that $1 \leq k \leq n-1$, I_k encloses only a loop statement with iterator I_{k+1}; otherwise it is imperfect. Although the assumption of perfect nesting is algorithmically convenient, it does exclude a considerable amount of application code. How to transform arbitrary imperfectly nested loops is still an open issue. In later chapters we show how the loop model can be extended to include some imperfect nestings.

Vector $\vec{I} = (I_1, \ldots, I_n)^T$ is called the *iteration vector*. It spans the set of all loop iterations represented by the *iteration space*, defined as follows:

Definition 1 (Iteration space) *The integer space,*

$$\mathcal{I} = \{(i_1, \ldots, i_n) \mid L_1 \leq i_1 \leq U_1, \ldots, L_n(i_1, \ldots, i_{n-1}) \leq i_n \leq U_n(i_1, \ldots, i_{n-1})\} \subseteq \mathbf{Z}^n,$$

is referred to as the iteration space, where $i_1, \ldots, i_n$ are the iteration indices, and $(L_1, U_1), \ldots, (L_n, U_n)$ are the respective loop limits. □

The iteration space is a convex polyhedron. The individual iterations in the iteration space are

[1] Nested loops can be normalized to have a unit stride [56], and such a transformation is illustrated in Appendix A.1.

```
for I1 = 0,5
   for I2 = 0,5
      A(I1,I2) = A(I1 - 1,I2) + A(I1,I2 - 1) + A(I1 - 2,I2 + 1)
   end for
end for
```

Figure 2.2: Nested loop L.

denoted by integer n-tuples, and we define a lexicographical relation on these n-tuples.

Definition 2 (Lexicographical relation $<$) *The lexicographical relation, $<$, is defined such that $\vec{i} < \vec{j}$ for n-tuples $\vec{i} = (i_1, \ldots, i_n)$ and $\vec{j} = (j_1, \ldots, j_n)$ iff there exists an integer k, $1 \le k \le n$ such that $i_1 = j_1, \ldots, i_{k-1} = j_{k-1}$ and $i_k < j_k$.* □

The lexicographical relation imposes an order on the iterations called the lexicographical order. This lexicographical order is the sequential execution order on the loop iterations.[2] A linear loop transformation, in effect, changes the lexicographical ordering of the iterations.

We will use the nested loop L of Figure 2.2 as a running example to illustrate the techniques in the linear loop transformation framework. The loop L has a dimension of two, where I_1 and I_2 are the iterators and $\vec{I} = (I_1, I_2)^T$ is the iteration vector. The loop limits are constants — both I_1 and I_2 have a lower limit of 0 and an upper limit of 5. Figure 2.3 on page 16 shows the iteration space for loop L.

2.1.2 Representation of Array References

A typical reference to an m-dimensional array A in the loop body has the form $A(f_1(I_1, \ldots, I_n), \ldots, f_m(I_1, \ldots, I_n))$, where each f_i is a function of the iterators and is called a subscript function. When the subscript functions are linear, then the references can be represented in matrix-vector notation. A reference to a p-dimensional array in an n-dimensional loop can be represented as a $p \times n$ integer matrix R, called a *reference matrix*. The reference matrix is typically extended to $p \times (n+1)$ to allow constant offsets in array subscript functions, where the last column of the reference matrix corresponds to the offsets in subscript functions. The iteration vector is correspondingly extended to $\vec{I} = (I_1, \ldots, I_n, 1)^T$ in this case. For example, array reference $A(I_1 - 2, I_2 + 1)$ in loop L is

[2] Note that the symbol $<$ is used to compare numbers as well as to compare tuples. Although this may be some what confusing at first, it simplifies the notation greatly, and the intended meaning should be clear from the context.

represented by:

$$\begin{bmatrix} 1 & 0 & -2 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} I_1 \\ I_2 \\ 1 \end{bmatrix}$$

and a reference $A(I_1 + I_2, I_2 + 1)$ with coupled subscript functions would be represented with:

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

Note that the reference matrix may, in some cases, be singular, such as for array reference $A(I_1, I_1)$.

2.1.3 Representation of Loop Bounds

The iteration space is a convex polyhedron and, as is evident from the definition of the iteration space, is characterized by the set of inequalities corresponding to loop bound expressions. The loop bound expressions can be written as inequalities in matrix-vector notation, since the bound expressions are linear functions of the iterators.[3] These inequalities are represented by half spaces that bound the iteration space, and are succinctly represented by a *bound matrix*. The set of lower bound expressions can be represented by

$$S_L \vec{I} \geq \vec{l}$$

where S_L is an $m_L \times n$ lower triangular integer matrix, m_L is the number of expressions bounding the loops from below, $\vec{I}$ is the $n \times 1$ iteration vector, and $\vec{l}$ is an $m_L \times 1$ integer vector. Similarly, upper bound expressions can be represented by

$$S_U \vec{I} \leq \vec{u} \quad \text{or} \quad -S_U \vec{I} \geq -\vec{u}$$

where S_U is an $m_U \times n$ upper triangular integer matrix and m_U is the number of expressions bounding the loops from above. The $m = m_U + m_L$ inequalities corresponding to both the upper and lower bounds can be combined to represent the polyhedral shape of the iteration space by:

$$S \vec{I} \geq \vec{c}$$

[3]The nested loops with a structure as in Figure 2.1 are called *affine loops*, when the array references and the loop bounds can be represented in a matrix-vector notation.

where

$$S = \begin{bmatrix} S_L \\ \hline -S_U \end{bmatrix}, \quad \vec{c} = \begin{bmatrix} l \\ -u \end{bmatrix}.$$

S is called the bound matrix. (where n is the loop dimension). When there are no maximum or minimum functions in the loop bound expressions, then the number of inequalities, m, is $2n$. Otherwise, m will be greater than $2n$. For example, if the lower bound expression for loop index I_2 is $\max(I_1, 5 - I_1)$, then $I_2 - I_1 \geq 0$ and $I_2 + I_1 \geq 5$ both belong to the set of inequalities S_L. Sometimes it is convenient to represent the loop bounds in the homogeneous co-ordinate system by inequalities $S\vec{I} \geq \vec{0}$, where S includes the vector $\vec{c}$ as the last column.[4]

As an example, consider the loop bounds for our running example loop L in matrix-vector notation. From the upper and lower loop bound expressions of loop L we can identify:

$$S_L = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \vec{l} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad S_U = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \vec{u} = \begin{bmatrix} 5 \\ 5 \end{bmatrix}$$

Thus the loop bounds can be represented by

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ \hline -1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} I_1 \\ I_2 \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ \hline -5 \\ -5 \end{bmatrix}$$

The constants on the right hand side can be incorporated into the bound matrix by using the homogeneous co-ordinate system. For instance, the loop bounds for L can now be represented as:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \hline -1 & 0 & 5 \\ 0 & -1 & 5 \end{bmatrix} \begin{bmatrix} I_1 \\ I_2 \\ 1 \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ \hline 0 \\ 0 \end{bmatrix}$$

The bound matrix represents the four half spaces that bound the two dimensional iteration space. In this particular case, the half spaces are $I_1 \geq 0$, $I_2 \geq 0$, $5 - I_1 \geq 0$, and $5 - I_2 \geq 0$.

[4] In a homogeneous co-ordinate system, the iteration vector $(I_1, \ldots, I_n)^T$ is extended to be $(I_1, \ldots, I_n, 1)^T$ so that linear equalities and inequalities in the system need not have non-zero constant terms on the right hand side.

2.2 Data Dependence Analysis

Determining the precedence constraints on the execution of the statements of a program is a fundamental step in parallelizing a program. The dependence relation between two statements constrains the order in which the statements may be executed. For example, the statements in the **then** clause of an **if** statement is *control dependent* on the branching condition. A statement that uses the value of a variable assigned by an earlier statement is *data dependent* on the earlier statement[8]. In this chapter, we concern ourselves only with data dependence.[5] We briefly discuss the basic concept of data dependence and the computational complexity of deciding whether a dependence exists or not. See Banerjee for a good reference of early development in the area [8]. Recent developments can be found in [37, 40, 45, 57].

There are four types of data dependences: *flow, anti, output,* and *input* dependence. The only *true dependence* is the flow dependence. The other dependences are the result of reusing the same location of memory and are hence called *pseudo dependences.* They can be eliminated by renaming some of the variables [14]. We write $S_1 \delta S_2$ to mean flow dependence from S_1 to S_2.

Definition 3 (Flow Dependence) *An iteration $\vec{j} \in \mathcal{I}$ is flow dependent on iteration $\vec{i} \in \mathcal{I}$, denoted $\vec{i}\delta\vec{j}$, iff there exists an element of an array assigned to in $\vec{i}$ and referenced in $\vec{j}$, such that $\vec{i} < \vec{j}$ and there is no $\vec{k}$, $\vec{i} < \vec{k} < \vec{j}$ with $\vec{k}\delta\vec{j}$ for the same array element.*

The sequential semantics of a loop implies that a flow dependence always be positive so that the array element in question is written to before the dependent iteration reads it. A loop transformation is *valid* or *legal* only if the flow dependences remain positive.

Determining the dependences in a loop, however, is a computationally hard problem. Two references to an m-dimensional array A, $A(f_1(\vec{i}), \ldots, f_m(\vec{i}))$ and $A(g_1(\vec{j}), \ldots, g_m(\vec{j}))$ access the same array element iff $f_1(\vec{i}) = g_1(\vec{j}), \ldots, f_m(\vec{i}) = g_m(\vec{j})$ for iterations $\vec{i}$ and $\vec{j}$. For conciseness, we denote this set of equalities by $F(\vec{i}) = G(\vec{j})$. The dependence problem can then be stated as follows: Do there exist iterations $\vec{i'}$ and $\vec{j'}$ in the iteration space such that $F(\vec{i'}) = G(\vec{j'})$? In other words, we need to know whether the following integer programming problem has integer solutions:

$$S\vec{i} \geq \vec{c}$$

[5] Control dependence is important to identify functional level parallelism, and to choose between various candidates for data distribution, among other things. Since, our discussion is limited to data flow analysis between loop iterations, control dependence does not concern us much.

$$S\vec{j} \geq \vec{c}$$

$$F(\vec{i}) = G(\vec{j})$$

This problem is NP-complete [20]. Moreover, for restructuring purposes we need more information than the mere existence of a dependence. We often need to know all pairs of iterations that are dependent and the precise relationships between them. An explicit representation of the dependences between all pairs of iterations would be tedious, space consuming and hard to use. Luckily, the structure of the dependences between all pairs of iterations of a nested loop is very regular because we are dealing with perfectly nested loops. If there is a dependence between iterations i and $i+k$, then there will also be a dependence between iterations j and $j+k$. Therefore, all the dependences can be concisely represented by a small set of dependence distance vectors.

Definition 4 (Dependence Distance Vector) *For a pair of dependent iterations $\vec{i} = (i_1, \ldots, i_n)^T$ and $\vec{j} = (j_1, \ldots, j_n)^T$ such that $\vec{i}\delta\vec{j}$, the vector $\vec{j} - \vec{i} = (j_1 - i_1, \ldots, j_n - i_n)^T$ is called the dependence distance vector.*

A dependence vector, say (a, b) in a 2-dimensional loop, represents dependences between all pairs of iterations $(i_1, i_2)\delta(i_1 + a, i_2 + b)$. For example, in our running example, loop L, iteration (I_1, I_2) is involved in 6 dependences: $[A(I_1, I_2), A(I_1 - 1, I_2)]$, $[A(I_1, I_2), A(I_1, I_2 - 1)]$, $[A(I_1, I_2), A(I_1 - 2, I_2 + 1)]$, $[A(I_1 + 1, I_2), A(I_1, I_2)]$, $[A(I_1, I_2 + 1), A(I_1, I_2)]$ and $[A(I_1 + 2, I_2 - 1), A(I_1, I_2)]$. In total, there are 80 dependences in the iteration space as shown in Figure 2.3. All of these dependences can be represented by 3 dependence distance vectors, namely $(1, 0)$, $(0, 1)$ and $(2, -1)$.

A dependence distance vector is called *constant* or *uniform* if the elements in the vector are constants. For example, $(1, 2)$ is a constant dependence[6]. In contrast, $(i, 0)$, is a *linear dependence*. The dependence distances in the examples of this chapter are generally uniform, so for conciseness, we refer to a uniform dependence distance vector as simply a dependence.

A vector of symbols $+$, $-$ and 0 corresponding to whether the elements of the dependence vector are positive, negative or zero is called a direction vector. The dependence direction vectors for our running example are $(+, 0)$, $(0, +)$ and $(+, -)$.

A dependence $(d_1, \ldots, d_n)$ is positive if the first non-zero element d_k is positive, and we say that the dependence is carried by the kth loop. The dependence in the loop are represented by the *dependence matrix*, D, an $n \times m$ integer matrix, where n is the dimension of the nested loop and

[6]In order to improve readability, we will use row vectors to represent dependence distance vectors, whenever possible.

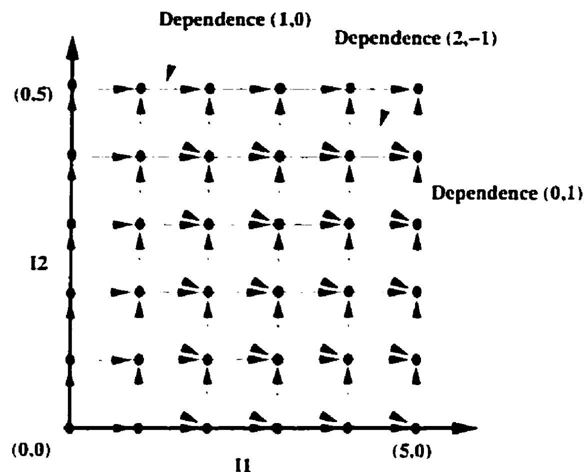


Figure 2.3: Dependences in iteration space.

m is the number of dependences. That is, each column of D corresponds to a dependence D_i. The matrix for loop L is

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

A dependence matrix is lexicographically positive, when all the dependences in the matrix are lexicographically positive.

Because the problem of determining the existence of a dependence is NP-complete, one often employs a more efficient algorithm in practice that solves restricted cases or provide approximate solutions. The GCD test, for example, finds the existence of an integer solution to the dependence problem with only a single subscript in an unbounded iteration space [8]. Some algorithms find real valued solutions in bounded iteration spaces with dependence direction information [37, 57]. However, Pugh recently noted that integer programming solutions, with exponential worst case complexity have much lower average complexity [45]. He modified the *Fourier-Motzkin* variable elimination technique [49] to produce exact solutions in a reasonable amount of time for many, but not all, problems.

2.3 Mathematics of Linear Loop Transformations

In this section, we describe the basic techniques of applying linear transformations to perfectly nested affine loops, including methods to derive the new dependences, references and loop bounds.

2.3.1 Basic Transformation Technique

A linear transformation of an n-dimensional perfectly nested affine loop is defined by an $n \times n$ non-singular integer matrix U, which maps the iteration space of the original loop onto a new iteration space so as to define a new loop nest. As a result of this transformation, the original iteration vector $\vec{I}$ is changed to a new iteration vector $\vec{K} = (K_1, ...K_n)^T$.

$$U\vec{I} = \vec{K} \tag{2.1}$$

and each dependence vector D_i in D is changed to a new vector D'_i:

$$UD_i = D'_i \tag{2.2}$$

The transformation U is legal iff each of the D'_i is lexicographically positive. The positivity of the transformed dependences is a necessary and sufficient condition for the legality of the transform.

As an example, consider a linear transformation of loop L. Suppose transformation U is applied to L, where

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

Then, we have

$$K_1 = I_1 + I_2, \; K_2 = I_2$$

The dependence matrix D for loop L,

$$D = \{(1, 0), (0, 1), (2, -1)\}$$

is changed to a new dependence matrix D':

$$D' = \{(1, 0), (1, 1), (1, -1)\}$$

Note that all of the transformed dependences are lexicographically positive, so U is a legal transformation.

A linear transformation maps each iteration in the original iteration space onto a new point in the new iteration space. Hence, a linear transformation only modifies the ordering of when

an iteration is executed relative to other iterations; it does not change what is computed in an iteration. Thus, with a linear transformation U, both an iteration $(i_1, \ldots, i_n)^T$ of the original iteration space, and the corresponding iteration $U(i_1, \ldots, i_n)^T$ of the new iteration space access the same set of array elements. Therefore, a reference matrix R in the original loop is modified to be matrix $R' = RU^{-1}$, since $R\vec{I} = RU^{-1}U\vec{I} = RU^{-1}\vec{K}$. For example, the array reference $A(I_1 + I_2, I_2)$ has a reference matrix of $\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$ which is modified to be $\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, since $U^{-1} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}$. The the new array reference thus becomes $A(K_1, K_2)$.

2.3.2 Derivation of New Loop Bounds

The bounds of the original loop are represented by inequalities:

$$S\vec{I} \geq \vec{0}$$

which is equivalent to

$$S\,U^{-1}\,U\vec{I} \geq \vec{0}.$$

Since $U\vec{I}$ can be replaced by $\vec{K}$, we have

$$S\,U^{-1}\,\vec{K} \geq \vec{0} \tag{2.3}$$

which is a set inequalities in terms of the new iteration vector. This set describes a convex polyhedron, and the sought after loop bounds are the set of integral, *affine* functions that bound the polyhedron.[7] The new bounds matrix is $S' = SU^{-1}$. The new loop bounds can be obtained directly from the rows of S', when each row of S' contains a single non-zero integer. If this is not the case, but the original loop bounds are simple (i.e., constants or simple functions of iterators), then it is possible to analytically derive symbolic expressions for the new loop bounds in terms of the original loop bounds and the elements of the transformation matrix [10, 28]. However, these symbolic expressions tend to be complex when the loop has more than three dimensions or has loop limits that are not constants.

The most general way of deriving the bounds from S' is to apply a variable elimination technique

[7] An integral, affine function can be represented by $\vec{w} \cdot \vec{I} = c$, where $\vec{w}$ is an integer vector, $\vec{I}$ is the iteration vector and c is an integer.

such as Fourier-Motzkin [47, 49]. This variable elimination technique derives the bounds for the new iterators inside out, that is K_n to K_1. It first rearranges the inequalities of S' such that K_n is alone on one side of inequalities. This rearrangement results in two sets of inequalities: in one set, K_n is more than or equal to an expression in iterators $K_1, \ldots, K_{n-1}$ i.e. $K_n \geq \alpha_1/c_1, \ldots, K_n \geq \alpha_l/c_l$ (where α_i's are linear expressions in $K_1, \ldots, K_{n-1}$ and c_i's are non-zero constants), and in the other set K_n is less than or equal to an expression in iterators $K_1, \ldots, K_{n-1}$, i.e. $K_n \leq \beta_1/b_1, \ldots, K_n \leq \beta_u/b_u$ (where β_i's are linear expressions in $K_1, \ldots, K_{n-1}$ and b_i's are non-zero constants). The first set of inequalities defines the lower bound for K_n, namely $max(\lceil \alpha_1/c_1 \rceil, \ldots, \lceil \alpha_l/c_l \rceil)$ and the second set defines the upper bound for K_n, namely $min(\lfloor \beta_1/b_1 \rfloor, \ldots, \lfloor \beta_u/b_u \rfloor)$.

At this point, K_n can be eliminated from the two sets of inequalities by considering the $l \times u$ pairs of inequalities from the two sets. An inequality $K_n \geq \alpha_i/c_i$ from the first set and an inequality $K_n \leq \beta_j/b_i$ from the second set can be combined into $\alpha_i/c_i \leq K_n \leq \beta_j/b_j$, where $1 \leq i \leq l$ and $1 \leq j \leq u$, after which K_n can be dropped to obtain $\alpha_i/c_i \leq \beta_j/b_j$ or simply $b_j\alpha_i \leq c_i\beta_j$. These new inequalities without K_n can then be used to derive the bounds for K_{n-1} as we did for K_n.[8] The process is then applied to eliminate $K_{n-2}, \ldots, K_1$, in that order. The bounds for K_1 will be constants.

In order to illustrate the derivation of loop bounds using the Fourier-Motzkin variable elimination technique, consider the transformation of loop L by the transformation matrix $U = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$. The loop bounds for loop L are represented by inequalities:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ \hline -1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} I_1 \\ I_2 \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ \hline -5 \\ -5 \end{bmatrix}$$

The transformation by U modifies the inequalities to be:

$$\begin{bmatrix} 1 & -1 \\ 0 & 1 \\ \hline -1 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} K_1 \\ K_2 \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ \hline -5 \\ -5 \end{bmatrix}$$

[8] Note that some of the new inequalities may be redundant.

We apply the variable elimination method in the following way. From the new set of inequalities, it is clear that,

$$K_1 - K_2 \geq 0, \quad K_2 \geq 0, \quad -K_1 + K_2 \geq -5, \quad -K_2 \geq -5$$

which are rearranged to obtain inequalities:

$$K_2 \leq K_1, \quad K_2 \geq 0, \quad K_2 \geq K_1 - 5, \quad K_2 \leq 5$$

From these inequalities we obtain the bounds for K_2 to be:

$$K_2 \geq max(K_1 - 5, 0) \text{ and } K_2 \leq min(K_1, 5)$$

or

$$0 \leq K_2 \leq 5 \text{ and } K_1 - 5 \leq K_2 \leq K_1$$

from which we obtain

$$0 \leq K_2 \leq K_1 \text{ and } K_1 - 5 \leq K_2 \leq 5$$

K_2 is then eliminated from these inequalities to obtain four inequalities involving only K_1:

$$0 \leq 5, \quad K_1 \leq K_1 + 5, \quad 0 \leq K_1, \quad K_1 - 5 \leq 5$$

Ignoring the first two redundant inequalities, we obtain constant bounds for K_1:

$$K_1 \geq 0 \text{ and } K_1 \leq 10$$

Figure 2.4 depicts the bounding lines for the transformed iteration space for loop L. The transformed loop L is shown in Figure 2.5.

A unimodular transformation matrix ensures that the inverse of the transformation matrix is also integer and unimodular. A unimodular transformation, therefore, maps from the original iteration space to the new iteration space one to one and onto. That is, every integer point in the original iteration space is mapped onto a unique integer point in the new iteration space and vice versa. Therefore, both the original and the transformed loops have unit stride.

When the transformation matrix is non-unimodular, then the inverse transformation matrix is not integer. In such a case, the transformed iteration space does not correspond to the original

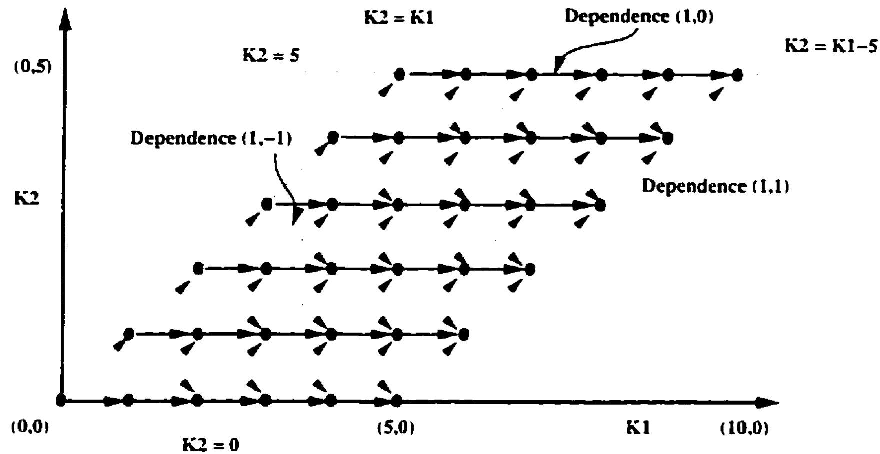


Figure 2.4: The transformed iteration space of loop L.

$$
\begin{aligned}
&\textit{for } K_1 = 0, 10 \\
&\quad \textit{for } K_2 = max(0, K_1 - 5), min(5, K_1) \\
&\qquad A(K_1 - K_2, K_2) = A(K_1 - K_2 - 1, K_2) + A(K_1 - K_2, K_2 - 1) \\
&\qquad\qquad\qquad + A(K_1 - K_2 - 2, K_2 + 1) \\
&\quad \textit{end for} \\
&\textit{end for}
\end{aligned}
$$

Figure 2.5: The transformed loop L.

iteration space exactly, since some integer points of the new iteration space map to non-integer points in the original iteration space. Therefore, it is necessary to have non-unit strides in the transformed loop so that it executes only those iterations which correspond to iterations present in the original loop. The strides for the transformed loop can be derived from the transformation matrix itself by simple matrix operations [47].

2.4 Advantages of Linear Loop Transformations

The linear loop transformation framework systematizes the task of a restructuring compiler in i) representing nested loops and their transformations, ii) applying loop transformations, iii) reasoning about the effects of the transformations, and iv) automatically deriving transformations that achieve desired effects. In this section, we briefly discuss the first three aspects: the next section describes techniques to derive linear loop transformations.

Firstly, the linear loop transformation framework provides a unified view of numerous transformations that had existed prior to the introduction of the linear loop transformation framework.

$$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \quad \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$$
$$(a) \qquad\qquad (b) \qquad\qquad (c)$$

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 & p \\ 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 \\ p & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix}$$
$$(d) \qquad\qquad (e) \qquad\qquad (f) \qquad\qquad (g)$$

Figure 2.6: Example transformation matrices for two dimensional loops, including (a) reversal of outer loop, (b) reversal of inner loop, (c) reversal of both loops, (d) interchange, (e,f) skew by p in second and first dimensions, and (g) wavefront respectively.

Before the framework was introduced, a restructuring compiler typically searched for an appropriate sequence of loop transformations such as loop interchange, permutation, skew, reversal, wavefront, and tiling.[9] With the linear transformation framework, a single transformation matrix can represent many of these loop transformations. For example, the transformation matrix for a given permutation is just a permuted identity matrix. A transformation matrix that reverses the kth loop level is an identity matrix with k^{th} row multiplied by -1. Figure 2.6 shows matrices for some important transformations of two dimensional loops.

With the linear transformation framework, a single transformation matrix can also represent any compound application of these transformations. For instance, the compound transformation of two loop transformations T_1 followed by T_2 is simply equal to their product: $T = T_2T_1$. Producing a transformed loop with a single matrix representing a transformation sequence is more efficient than producing the transformed loop as a sequence of individual transformations. With a single matrix, the expressions in array references and loop bounds tend to be simpler and thus more efficient to compute, since the transformed loop structure is derived only once for the entire transformation sequence.

Secondly, the linear algebra framework enabled the design of generic techniques to derive the loop bounds, references and dependences of the transformed loop irrespective of the actual transformation being applied. As described in Section 2.5, these derivations generally involve only simple linear algebra so they can easily be performed automatically by a compiler.

Finally, the linear loop transformation framework makes it easier to characterize aggregate effects of transformation sequences, so as to reason about the "goodness" of a sequence. This

[9] Appendix A provides details on these and several other loop transformations.

makes it possible to design heuristic algorithms capable of automatically deriving transformations given specific optimization objectives [26, 29, 36, 48, 54]. The performance aspects of a nested loop — such as parallelism at outer (inner) loop level, volume of communication, the average load, and load balance — can be characterized in terms of the elements of the transformation matrix, dependences, and original loop bounds [26]. As an example, consider the transformation of a nested loop to maximize inner loop parallelism by transforming the loop so that the new dependences are independent of the inner loop levels. A loop has inner loop parallelism whenever the first element of each dependence vector is positive (non-zero) [10, 31]. Hence we require a transformation U such that the first element of all transformed dependence vectors are positive (non-zero). Any transformation U such that

$$\forall \vec{d_i} \in D, \; \vec{U_1} \cdot \vec{d_i} > 0$$

where U_1 is the vector corresponding to the first row of U, is a desired transformation. It can be shown that such a transformation exists for all perfectly nested loops [10, 31].

As a second example, consider the synchronization overhead in a nested loop with only inner loop parallelism. The number of synchronizations is proportional to the number of iterations of the outermost loop. For any candidate transformation U, the size of the outer loop is characterized by the difference between the maximum and the minimum possible values for iterator K_1:

$$max(\vec{U_1} \cdot \vec{I}) - min(\vec{U_1} \cdot \vec{I}) + 1, \; \vec{I} \in \mathcal{I}$$

where $\vec{U_1}$ is the vector corresponding to the first row of U (since $K_1 = U_1 \cdot \vec{I}$) [26]. This expression provides an objective function that can be used to evaluate the goodness of candidate transformations.

As a final example, consider loop transformations to parallelize the outermost loop level. A loop nest has outer loop parallelism if the first element of each transformed dependence vector is 0. A candidate transformation, U, parallelizes the outermost level if

$$\forall \vec{d_i} \in D, \; \vec{U_1} \cdot \vec{d_i} = 0 \text{ and } U\vec{d_i} \geq \vec{0}$$

where $\vec{U_1}$ is the vector corresponding to the first row of U. That is, a transformation U parallelizes the outer loop if $\vec{U_1}$ is orthogonal to every dependence vector. The goodness of a candidate transformation is also determined by the number of parallel iterations in the transformed loop. The objective is, therefore, to select a transformation U which parallelizes the outermost loop level and maximizes:

$$max(\vec{U_1} \cdot \vec{I}) - min(\vec{U_1} \cdot \vec{I}) + 1, \vec{I} \in \mathcal{I}.$$

2.5 Techniques to Derive Linear Loop Transformations

The derivation of a linear transformation that is optimal for a given optimization objective is, unfortunately, hard in general. The problem is NP-complete for unrestricted loops and even affine loops with non-constant dependence distances [15]. However, approximate solutions can be derived efficiently by using the desired properties of the transformed loop to guide the search for a transformation. For example, transformations to parallelize outer loops of a perfectly nested affine loop can be derived in polynomial time by using the dependence matrix to guide the search [28, 29]. Typically, such algorithms apply a sequence of matrix row operations to transform the original dependence matrix into a new dependence matrix that has only zeros in the first (or first few) row(s). Each matrix row operation that is applied to the dependence matrix is also simultaneously applied to a matrix that was originally started out as an identity matrix. The sequence of operations on the identity matrix results in the transformations matrix. An important property of such matrix manipulation based techniques is that the existence proofs tend to be constructive. That is, the existence proofs entail algorithms that derive the sought after transformation.

The following subsections describe how a desired property of the transformed loop structure can be used to derive a linear loop transformation in four different optimization contexts.

2.5.1 Deriving Canonical Loops

A nested loop is said to be in canonical form when a maximum number of its outermost loop levels are fully permutable [55]. A loop nest in canonical form has the advantage that its outer loop levels can be permuted or skewed in any way to suit the target architecture. For example, loop levels can be permuted to bring parallelism to the outermost level and the loop nest can be tiled to improve cache locality.

Algorithms to derive matrices that transform loop nests into their canonical forms use the dependence matrix to guide the search, because the structure of the dependence matrix characterizes full permutability. For the loop levels i to j of a loop nest to be fully permutable, the elements $d_i, \ldots, d_j$ of each column $(d_1, \ldots, d_n)^T$ of the dependence matrix must all be positive [9]. The algorithm by Wolf and Lam incrementally derives a list of outer fully permutable loop levels [55]. Starting with an empty list, the algorithm adds a loop level to the list by making the loop level

```
for  i = 0, n                          for  K1 = -n, n
  for  j = 0, n                          for  K2 = max(0, -K1), min(n, n - K1)
    A(i, j) = A(i - 1, j - 1)     ==>      A(K1 + K2, K2) = A(K1 + K2 - 1, K2 - 1)
  end for                                end for
end for                                end for
```

Figure 2.7: Dependence internalization in two dimensional loop.

permutable using permute, reverse and/or skew transformations. The authors argue that their algorithm requires minimal number of these transformations when the the loop dimension is less than five [55].

Algorithm to derive transformations for dependence *internalization*, described next, can be adapted as an alternative approach to derive transformations to obtain loops in canonical form.

2.5.2 Dependence Internalization

Dependence *internalization* transforms a loop nest so that a maximum number of dependences are independent of the outer loop levels [26, 28]. They are designed to extract coarse grain parallelism in nested loops. The algorithms to derive dependence internalization transformations also use the dependence matrix to guide the search: in fact, the transformation matrix is constructed by systematically manipulating the dependence matrix. The general framework for internalization and algorithms to find a good internalization in polynomial time are given by Kumar et al. [28, 29].

As an example, consider the loop on the left hand side of Figure 2.7. Its $(1, 1)$ dependence can be internalized to be $(0, 1)$, so that the K_1 loop level is parallel (i.e. has no dependences). This transformation modifies the original loop on the left to become the loop on the right hand side. One unimodular matrix U (of several) that achieves the above internalization of $(1, 1)$ is:

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

The algorithm to derive a dependence internalization of an n-dimensional nested loop consists of two steps [28]. In the first step, the dependence matrix is re-arranged so that $n-1$ selected dependences form the first $n-1$ columns.[10] The dependences are selected as follows. Each dependence $(d_1, \ldots, d_n)^T$ is treated as an integer point $(d_1, \ldots, d_n)$, and a convex hull of the chosen integer points and the origin is constructed. The dependences on a face of the convex-hull containing the origin are selected as the first $n-1$ columns. This ensures legality because a vector orthogonal to

[10] In order to simplify the presentation here, we assume that the number of dependences is greater than or equal to $n-1$.

these dependences can be the first row of the transformation matrix so that all the dependences have a non-negative dot product with the first row.[11] Thus, the first element of each dependence is either non-negative or zero.[12] [28].

As an example of the first step of the algorithm, consider dependences $(1,1)^T$, $(2,1)^T$ and $(1,2)^T$ in a 2-dimensional loop. The convex-hull of integer points (0,0), (1,1), (2,1), and (1,2) is a polygon, where $(1,2)^T$ and $(2,1)^T$ each lie on a line of the polygon that contains origin. Assuming we select $(1,2)^T$ as the first column of the dependence matrix, a perpendicular to $(1,2)^T$, namely $(2,-1)^T$, would eventually become the first row of the transformation matrix. Note that all the dependences have a non-zero dot product with $(2,-1)^T$. On the other hand, selecting $(1,1)^T$ as the first column of the dependence matrix would make the transformation illegal, since $(1,-1)^T$ would become the first row of the transformation matrix, which has negative dot product with dependence $(1,2)^T$.

In the second step, the dependence matrix is augmented with the n-dimensional identity matrix. The algorithm systematically applies matrix row operations to the augmented matrix to make the elements of as many rows of the dependence matrix as possible zeros. For instance, when all elements in the first row of the dependence matrix are zero, all dependences are internalized to second level, so that the outer loop is parallel.

The basic idea in the technique just described for deriving transformation for dependence internalization can also be employed to obtain canonical forms of nested loops. The derivation of the transformation matrix that makes a loop fully permutable has two steps. First, matrix row operations are used to make all elements of one row of the dependence matrix, say the last, positive.[13] This step involves dependence internalization followed by row operations to add multiples of the rows containing the first non-zero elements of the dependences to the last row until it has only positive elements. In the second step, appropriate multiples of the last row (which has only positive elements) are added to other rows that still have some negative elements.

2.5.3 Access Normalization

Access normalization [35] is a linear loop transformation to modify the order in which array elements are accessed so as to improve cache and memory access locality. Access normalization

[11] Note that the first row of the transformation matrix is not derived by finding the vector orthogonal to a hull-face. The second step of the algorithm automatically derives all rows of the transformation matrix, once the dependence matrix is appropriately re-arranged.

[12] The first elements will all be zero when the rank of the dependence matrix is less than n.

[13] This is possible since each dependence has a non-zero positive element.

```
for i = 0, N_1 - 1                                     for u = p.b - 1, P
  for j = i, i + b - 1                                   for v = u, u + N_1 + N_2 - 2
    for k = 0, N_2 - 1                                     for w = 0, N_1 - 1
      B(j - i, i) = B(j - i, i) + A(j + k, i)   ==>          B(u, w) = B(u, w) + A(v, w)
    end for                                                end for
  end for                                                end for
end for                                                end for
```

Figure 2.8: An example access normalization.

transformations are derived using the array subscript functions to guide the search. Cache locality is improved, when successive iterations access array elements that are either the same or spatially close (i.e. in the same cache line or in the same page). Memory locality is improved, when more of the data a processor must access is local so that fewer remote accesses are necessary. In order to improve memory locality, a target loop has to be transformed so that more of the array accesses in the iterations assigned to a processor are to those portions of the arrays mapped onto the processor.

As an example of improving memory locality, consider the loop on the left hand side of Figure 2.8. Suppose that the arrays A and B are distributed onto the processors by rows, and that each iteration of the outer loop is executed in parallel. A processor executing an outer loop iteration needs to access a new row of B in each j iteration and a new row of A in every iteration. This invariably results in an excessive number of non-local accesses. The number of remote accesses can be reduced substantially by transforming the loop so that each outer loop iteration accesses mostly the local rows of the arrays. In order to achieve this, the transformed loop should be such that the row array subscripts of the references are simple functions of the parallel loop iterator.

Such a transformation matrix can be derived by first constructing a matrix, called the access matrix, where the rows correspond to the subscript functions of the array references. For instance, expressions $j - i$, $j + k$, and i in the loop on the left hand side of Figure 2.8 are represented by the access matrix-vector pair:

$$\begin{bmatrix} -1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} = \begin{bmatrix} j - i \\ j + k \\ i \end{bmatrix}$$

The representation is similar to the reference matrix in that the rows of the access matrix represent the subscript functions in array references. However, a reference matrix represents all subscript functions of the same array reference, where as an access matrix has selected subscript functions from several array references, potentially to different arrays.

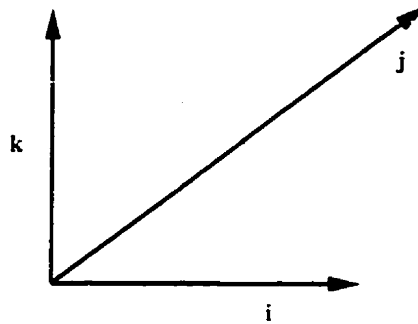


Figure 2.9: Choice of iterator to partition for good load balance.

The transformation matrix is then formed by selecting the first n rows of the access matrix (where n is the loop dimension). The access matrix is suitably padded when it has fewer than n rows. When the first n rows do not form a legal transformation, then a legal transformation matrix is formed by selecting fewer than n rows from the access matrix and suitably padding the transformation matrix. For the example of Figure 2.8, the access matrix happens to be a legal transformation matrix.

The access normalized loop is shown on the right hand side of Figure 2.8. As an effect of access normalization, $j - i$ became u and and $j + k$ became v. In the transformed loop, therefore, all accesses to B are local, although A still has some non-local accesses. Clearly, the number of non-local accesses is sensitive to the order of rows in the access matrix. A common heuristic to reduce the amount of communication required is to place in the first few rows of the access matrix the subscript functions in the array dimension(s) along which the array is distributed [35].

2.5.4 Balancing Processor Load

One of the major overheads associated with mapping parallel iterations onto a multiprocessor is load imbalance. Load imbalance is greatly influenced by the choice of the iterator that is parallel. For example, consider mapping the iterations of the 3-dimensional iteration space of Figure 2.9 onto a multiprocessor. Statically partitioning the loop along the i (or the k) loop dimension results in load imbalance, since an unequal number of iterations are assigned to each processor. Partitioning along the dimension j, on the other hand, achieves good load balance. To obtain a good load balance, it is necessary that an iterator l be selected such that i) the bounds of the other iterators are not a function l and ii) the bounds of l are not a function of other iterators. It is possible both to determine whether a loop nest can be transformed to have these two properties

or not, and if so to derive an appropriate loop transformation [42]. It should be noted, however, that load balancing could offset other optimization objectives such as maximizing parallelism or minimizing communication. Heuristics exist that are capable of identifying transformations that balance the objectives of maximizing parallelism, minimizing communication and balancing the load for perfectly nested loops with constant loop limits [26].

Chapter 3

Computation Decomposition and Alignment Framework

Many small make a great.
— *John Heywood: Proverbes. part i. chap. xi.*

3.1 Overview of CDA Transformation Framework

The main idea behind CDA is to linearly transform loops at a granularity that is finer than what the linear loop transformation framework allows. A linear loop transformation reorganizes computations in a nested loop at the granularity of iterations; each iteration in the original iteration space is mapped onto a new point in the new iteration space. Hence, a linear transformation does not affect what is computed in an iteration, but only when it is executed relative to other iterations. In contrast, CDA can map just a portion of an iteration to a new point in the new iteration space. Figure 3.1 shows how mapping at such fine granularity changes the composition of the iterations themselves.

The basic transformation technique in CDA *i*) partitions the iteration space into possibly several integer spaces and *ii*) linearly transforms each of these integer spaces by a different transformation matrix, and *iii*) fuses the transformed integer spaces to obtain a new iteration space. The first two steps are called *Computation Decomposition* and *Computation Alignment*, respectively.

As an example, consider the original loop on the left hand side of Figure 3.2. The iteration space of this loop can be partitioned into two integer spaces. The first corresponds to the array accesses due to reference $B(i,j)$, and the second corresponds to the assignment to $A(i,j)$ using the result from the first space and the subexpression $A(i,j) + B(i+1,j)$. The first space is applied a simple linear transformation so that the $B(i,j)$ reference becomes a $B(i+1,j)$ reference.[1] The second space is applied the identity transformation (and thus remains unchanged). The transformed integer spaces are then fused together to form a new iteration space. The composition of these new

[1] That is, $B(i,j)$ is accessed in iteration $(i-1,j)$ instead of in iteration (i,j).

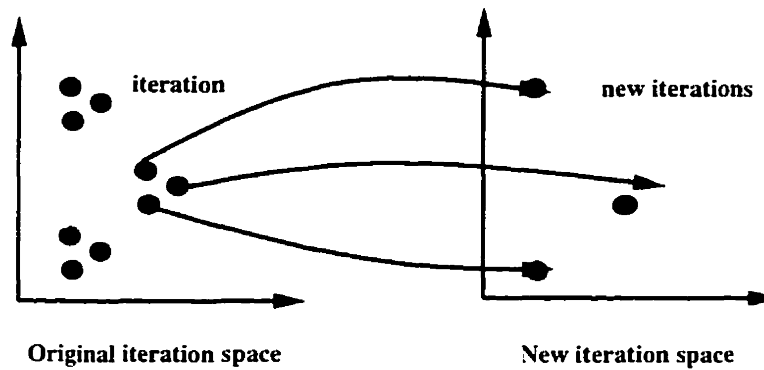


Figure 3.1: Mapping of portions of iterations in CDA. For clarity, this figure shows how the computations of a single original iteration are mapped onto new iterations.

```
for i = 1,n                          for i = 0,n
    for j = 1,n                          for j = 1,n
        A(i,j) = B(i,j) + A(i,j)+            if (i < n) then t(i + 1,j) = B(i + 1,j)
                 B(i + 1,j)        ==>       if (i > 0) then A(i,j) = t(i,j)+
    end for                                                      A(i,j) + B(i + 1,j)
end for                                  end for
                                     end for
```

Figure 3.2: An example CDA transformation.

iterations is different from the original iterations with regard to the array elements accessed and the computations performed. Hence this transformation could not have been achieved within the linear loop transformation framework. This particular CDA transformation improves cache hit rates in the loop for certain array sizes and target cache geometries (i.e., cache size, associativity and cache line size).

Because CDA transforms loops at a relatively fine granularity and rearranges the order in which instructions are executed, it is, in some ways, similar to instruction scheduling techniques that are common in today's optimizing compilers. Software pipelining [30] is such an instruction scheduling technique that moves instructions from one iteration to an earlier one so as to hide latencies and to improve instruction level parallelism. However, CDA differs from these instruction scheduling techniques in a number of very fundamental ways. First, CDA typically transforms code at a coarser granularity than the instruction scheduling transformations of single instructions, because CDA is a source level transformation. Second, CDA can map computations onto any point in the new iteration space, whereas instruction scheduling techniques typically move instructions within a basic block or to the previous execution(s) of the innermost loop. Third, the formal frameworks

underlying CDA and instruction scheduling techniques are fundamentally different. Finally, CDA, and linear loop transformation techniques in general, are used to target higher level optimization objectives than instruction level parallelism.

The granularity at which computations are mapped in CDA can vary from subexpressions, to assignment statements, to conditionals, to loop statements, and to entire iterations. When the granularity is an iteration, then the CDA transformation is equivalent to a linear loop transformation, so CDA subsumes linear loop transformations. However, by being able to also transform loops at a finer granularity, CDA provides additional opportunities for optimization.

The CDA framework retains the elegance and advantages of linear transformations while enabling new code optimizations. However, relatively fine-grained restructuring in the CDA framework implies that deriving CDAs is more complex than deriving linear transformations. Therefore, heuristic algorithms are key to efficient derivation of CDA transformations. Also, a CDA transformed loop typically has more overheads than a linearly transformed loop, so techniques to minimize the overhead become necessary.

In this chapter, we describe the CDA transformation technique in detail. The following chapter describes techniques to minimize overheads in CDA transformed loops. Later chapters present heuristic algorithms to derive CDA transformations in specific optimization contexts.

3.2 Representation of the Loop Structure

The CDA framework can transform nested loops of the type shown in Figure 3.3. Each statement L_i can be either an assignment statement, a conditional statement, or a perfectly nested loop. As with the perfectly nested loops of Chapter 2, the loop bounds are expressed by l_i and u_i, which are integral, *affine* functions of the enclosing iterators.[2] The loops are normalized so that the step size is one. The arrays in the loop body are indexed by integral, affine functions of the enclosing iterators.

When the L_i are loop statements, then we assume that they are perfectly nested, and that the dimension of all L_i is the same, namely $(n-k)$ for some $n \geq k$. This allows for transformation of integer spaces of the same dimension.[3] This program model is quite general. When there is a single sub-nest L_1, then the loop is an n-dimensional perfect nest. If $n = k$, then L_i are all non-loop

[2] An integral, affine function can be represented by $\vec{w} \cdot \vec{I} = c$, where $\vec{w}$ is an integer vector, $\vec{I}$ is the iteration vector and c is an integer.

[3] The model of nested loops can be extended to the case where L_i have different dimensions by adding dummy loop statements which iterate once.

```
for I_1 = l_1, u_1
   for I_2 = l_2(I_1), u_2(I_1)
      ...
         for I_k = l_k(I_1, ... I_{k-1}), u_k(I_1, ... I_{k-1})
            L_1 : ...
            L_2 : ...
            ⋮
         end for
      ...
   end for
end for
```

Figure 3.3: The program model

statements, so that we have a k-dimensional perfect nest. If $k = 0$, then L_i and L_j are statements (loop or otherwise) with no common nesting. If $k \neq 0$ and $n > k$, we have an imperfectly nested loop, where perfectly nested sub-nests L_i are enclosed by a k-dimensional perfect nest.

In general, the iteration vector I_i for loop L_i is $(I_1, ..., I_k, I^i_{k+1}, ..., I^i_n)$, where $I^i_{k+1}, ..., I^i_n$ are the iterators inside L_i. The iteration vector specifies an integer point representing an iteration in the iteration space under consideration. The bounds of $I_1, ..., I_k, I^i_{k+1}, ..., I^i_n$ characterize the iteration space where each integer point corresponds to an execution of L_i's body. When the context of the loop under consideration is clear, the iteration vector is denoted simply by $\vec{I} = (I_1, ..., I_n)$ and the corresponding iteration space by $\mathcal{I}$. Also, when the use of homogeneous coordinate system is clear, $\vec{I}$ denotes $(I_1, ..., I_n, 1)$. As in the linear loop transformation framework, an array reference can be represented by an $m \times (n+1)$ reference matrix and the iteration vector.

In this chapter, we consider only perfectly nested loops. This allows us to focus exclusively on optimization opportunities in transforming statements and subexpressions at the same nesting level. Section 7.4 describes specific application of CDA to imperfect loop nests.

3.3 Computation Decomposition

Computation Decomposition is the first step in a CDA transformation. The main objective of Computation Decomposition is to partition the iteration space into possibly several integer spaces, each representing a space of computations of a granularity that can be smaller than an entire iteration. Since it is natural to do so, we first describe Computation Decomposition as a transformation of the loop body. We will follow this description with a formalization of Computation Decomposition as a partitioning of the iteration space.

The CDA framework transforms nested loops at the textual granularity of *loop fragments*, where a loop fragment is a portion of the loop body, such as a subexpression, a conditional, an assignment or loop statement, or the entire loop body. The objective of Computation Decomposition is to divide or decompose the loop body into several *chosen* loop fragments, so that each statement of the new loop body corresponds to a loop fragment. First, it divides the loop body into its individual statements and then may additionally decompose individual statements into new statements of finer granularity that together have the same semantics as the original statement that was decomposed.

We first focus on the decomposition of assignment statements. An assignment statement is decomposed by rewriting it as a sequence of smaller statements that accumulate the intermediate results and produce the same final result. Consider an assignment statement, S_j, in a loop body with at least one binary operator *op*:

$$S_j : w_j \leftarrow f_{j.1}(R_{j.1}) \ op \ f_{j.2}(R_{j.2})$$

where w_j denotes the left hand side (lhs) reference. $R_{j.1}$ and $R_{j.2}$ are the sets of references in subexpressions $f_{j.1}(R_{j.1})$ and $f_{j.2}(R_{j.2})$ respectively. The above statement can be decomposed into the following two statements to produce the same result :

$$S_{j.1} : t_j \leftarrow f_{j.1}(R_{j.1})$$
$$S_{j.2} : w_j \leftarrow t_j \ op \ f_{j.2}(R_{j.2})$$

where t_j is a temporary variable introduced to accumulate the intermediate result.

The choice of subexpressions that are to be elevated to the status of statements is a key decision in CDA optimization. As we will see in later chapters, the specific optimization objective being pursued influences this decision. We can repeatedly decompose a statement into possibly many statements, with the result of each new statement held in a different temporary variable. The temporary variables are typically organized in arrays of the dimension and size of the iteration space. This gives Computation Alignment, which follows Computation Decomposition, more opportunities for optimization than just using scalar temporary variables. In particular, it ensures that there are no output dependences on the temporaries to constrain the number of candidate Computation Alignments. (While the introduction of the temporaries adds overhead, this overhead can be substantially reduced by the optimizations discussed in Chapter 4.)

Optimizing compilers implicitly decompose a conditional statement `if (condition) then body` into two statements `t = condition` and `if (t) then body`. This decomposition allows for separation of computations for the condition and the body, which can be used to improve instruction

```
for i = 1, n
 for j = 1, n                         S1.1                      S1.2

  S1  : A(i,j)   = A(i-1,j-1) + B(i-1,j) + A(i-1,j) + A(i,j-1) + B(i,j+1)

  S2  : B(i,j-1) = A (i,j-1) + B(i,j)

 end for
end for
```

(a) The original loop and the program fragments

```
for i = 1, n
 for j = 1, n
  S1.1 : t(i,j)    = A(i-1,j-1) + B(i-1,j) + A(i-1,j)
  S1.2 : A(i,j)    = t(i,j) + A (i,j-1) + B(i,j+1)
  S2   : B(i,j-1) = A(i,j-1) + B(i,j)
 end for
end for
```

(b) Loop after Computation Decomposition

Figure 3.4: Running example and new loop body after Computation Decomposition.

scheduling by knowing the branching ahead of time. However, in this thesis, we focus only on the advantages of decomposing the assignment statements in the body of the conditional statements. A subnest of the loop body is decomposed by decomposing the body of the subnest.[4]

As an example of Computation Decomposition, consider the top half of Figure 3.4, which shows a 2-dimensional loop. We will use this loop as a running example to illustrate the CDA transformation technique. The bottom half of the figure shows the running example after Computation Decomposition. In this case, Computation Decomposition first decomposes the loop body into statements S_1 and S_2. Statement S_1 is further split and replaced by two statements, $S_{1.1}$ and $S_{1.2}$. The result of $S_{1.1}$ is stored in the temporary $t(i,j)$, which is then subsequently used by $S_{1.2}$ Although not a requirement, we have chosen to have the reference matrix of the temporary array be the same as that of the lhs of the original statement S_1. Note that the bounds of the loop do not change when it is decomposed.

We can now formalize Computation Decomposition as a partitioning of the iteration space into several integer spaces. A *computation* of a loop fragment, s, in an iteration of the loop is defined to be the execution of s in the iteration. The computation of s in iteration $(i_1, \ldots, i_n) \in \mathcal{I}_n$ of an n-dimensional loop is denoted by $c(i_1, \ldots, i_n; s)$. For example, the computation of L_1 in iteration

[4] We do not decompose the loop statement of the subnest itself, say `for i=1, u`, into multiple loop statements, say `for i=l1, u1` and `for i=l2, u2`, since the loops typically do not have opportunities for aligning one portion of the iteration space to another portion.

$(i_1, \ldots, i_k)$ of the loop of Figure 3.3 is denoted by $c(i_1, \ldots, i_k; L_1)$. When s is the entire loop body, then $c(i_1, \ldots, i_n; s)$ denotes a complete iteration. Also, for completeness $c(o, s)$ denotes the execution of s, where s does not have any enclosing loops.

Similar to an iteration space, we can now define an integer space of computations for a given loop fragment, called a *computation space*.

Definition 5 (Computation space) *The computation space of loop fragment s in the loop body of an n-dimensional loop, denoted by $CS(\mathcal{I}_n, s)$, is an integer space representing the set of all computations of s in iteration space $\mathcal{I}_n$*

$$CS(\mathcal{I}_n, s) = \{c(i_1, \ldots, i_n; s) \;\mid\; \forall\ (i_1, \ldots, i_n) \in \mathcal{I}_n\ \}$$

□

Computation spaces are convex polytopes similar to iteration spaces. The computation space for L_1 of Figure 3.3 is $CS(\mathcal{I}_k, L_1)$, where $\mathcal{I}_k$ is the k-dimensional iteration space. When L_1 itself is an $(n-k)$-dimensional nested loop, and s is a statement in its loop body then $c(i_1, \ldots, i_k, i^1_{k+1}, \ldots, i^1_n; s)$ is a computation and $CS(\mathcal{I}, s)$ is the corresponding computation space, which is n-dimensional. When the context is clear, the computation space of s refers to the entire set of computations due to all the enclosing iterators, so that it can be denoted simply by $CS(s)$. The objective of Computation Decomposition can now be defined as a partitioning of the iteration space into several computation spaces.

Definition 6 (Computation Decomposition) *A computation decomposition of an n-dimensional loop with loop body S is the creation of m computation spaces $CS(S_1), \ldots, CS(S_m)$, where for every iteration $(i_1, \ldots, i_n) \in \mathcal{I}_n$ the computation $c(i_1, \ldots, i_n, S)$ and the computations*

$$c(i_1, \ldots, i_n; S_1), \ldots, c(i_1, \ldots, i_n; S_m)$$

executed in order produce the same result. □

When $m = 1$ in the above definition, then $S_m = S$, so the computation space is the same as the iteration space. The bounds of each of the computation spaces are the same as those for the iteration space before applying Computation Alignment.

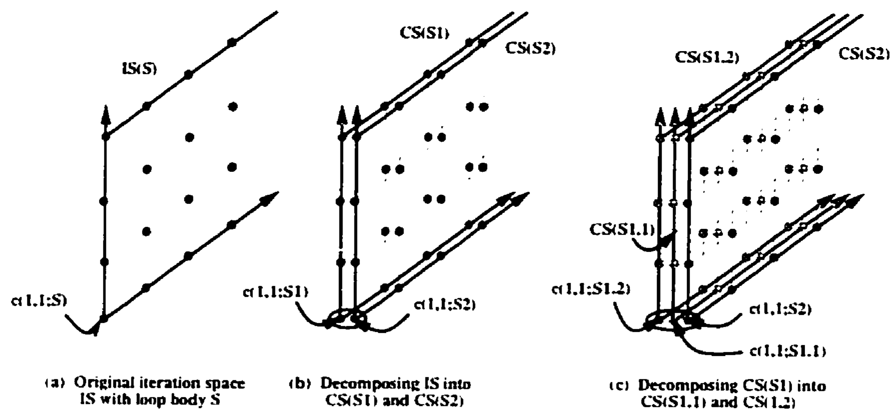


Figure 3.5: Computation spaces for the running example.

Figure 3.5 shows the decomposition of the running example in terms of the computation spaces. The iteration space is first decomposed into two computation spaces, one for each statement, as shown in Figure 3.5b; that is:[5]

$$CS(S) \equiv \{CS(S_1), CS(S_2)\}$$

In a second step, the computation space of S_1 is decomposed into two finer computation spaces, as shown in Figure 3.5c, so that:

$$CS(S) \equiv \{CS(S_{1.1}), CS(S_{1.2}), CS(S_2)\}$$

Although Computation Decomposition is a simple transformation, it is effective in exposing opportunities for fine grain restructuring of the loop.

3.4 Computation Alignment

Computation Alignment is the second step in a CDA transformation. Computation Alignment applies a separate linear transformation to each of the computation spaces. It modifies the loop bounds, dependences and array references, and the loop fragments that constitute the new loop body may be different from the loop fragments that constitute the original loop body.

Definition 7 (Computation Alignment) *A Computation Alignment of an n-dimensional nested loop with statements $S_1, \ldots, S_m$ in the loop body is the application of linear transformations $T_1, \ldots, T_m$*

[5] Here, we read $\equiv$ as "equivalent to".

to the computation spaces $CS(S_1), \ldots, CS(S_m)$, *respectively.* □

As in the linear loop transformation framework, the transformation matrices are integer and non-singular.

Constitution of the new iterations: Intuitively, Computation Alignment results in a relative movement of the individual computations across iterations. As a result, a new iteration may consist of computations that originally belonged to different iterations. With $(i_1, \ldots, i_n: S_j)$ denoting the computation of statement S_j in iteration $(i_1, \ldots, i_n)$, an iteration $(i_1, \ldots, i_n)$ in the original iteration space consisted of computations:

$$(i_1, \ldots, i_n) \equiv \{(i_1, \ldots, i_n: S_1), \ldots, (i_1, \ldots, i_n: S_m)\}$$

The corresponding iteration in the new iteration space consists of computations:

$$(i_1, \ldots, i_n) \equiv \{(i_1^1, \ldots, i_n^1: S_1), \ldots, (i_1^m, \ldots, i_n^m: S_m)\}$$

where

$$(i_1^j, \ldots, i_n^j)^t = T_j^{-1} \cdot (i_1, \ldots, i_n)^t$$

New array references: The new references within a given statement are derived as in the linear loop transformation framework: if computation space $CS(S)$ is transformed by transformation T, and r is a reference in statement S with reference matrix R, then r has a new reference matrix, $R \cdot T^{-1}$, after the transformation. The array accesses in the transformed loop may be fundamentally different from the accesses in the original loop, since all references are not modified using the same transformation matrix.

Consider the Computation Alignment of the running example. The three computation spaces in the decomposed loop can be computationally aligned by applying transformations

$$T_{1.1} = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad T_{1.2} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad T_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix}$$

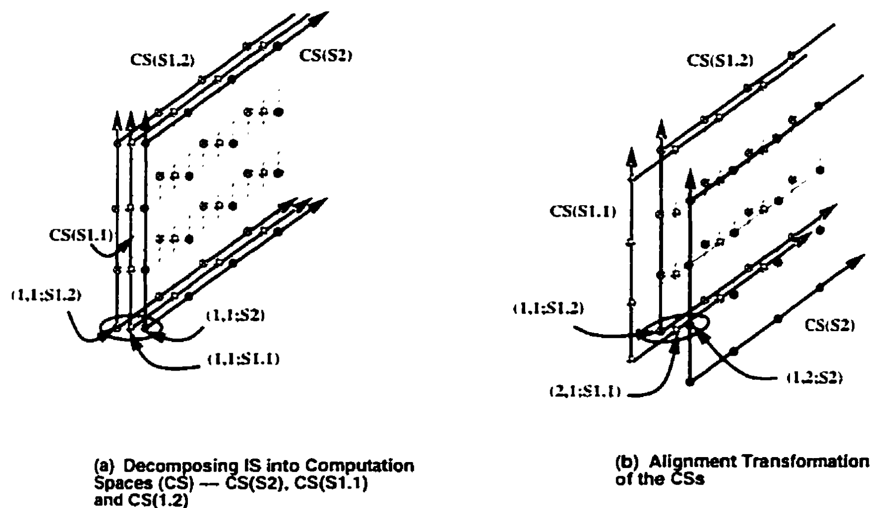


Figure 3.6: Illustration of a simple Computation Alignment of the computation spaces.

to computation spaces $CS(S_{1.1})$, $CS(S_{1.2})$ and $CS(S_2)$, respectively. These transformations are intended to align most references to $A(i,j)$. The computation spaces $CS(S_{1.1})$ and $CS(S_2)$ move relative to $CS(S_{1.2})$, since $T_{1.2}$ is the identity matrix. $CS(S_{1.1})$ moves one stride in direction i so that the $(i-1,*)$ references in $S_{1.1}$ change to $(i,*)$ references. This is shown in Figure 3.8. $CS(S_2)$ moves one stride in direction j so that the $B(i,j-1)$ reference changes to $B(i,j)$. Figure 3.6(b) shows the transformed computation spaces and highlights three computations that are now executed in one iteration.

New dependence relations: The CDA changes the dependence relations in the loop. When a dependence exists between two references within a statement of the decomposed loop body, then the new dependence can be derived as in the linear loop transformation framework, because both the read and write references in the statement are modified using the same transformation matrix. The dependence d between a write reference w and a read reference r of statement S is modified to be Td assuming the computation space $CS(S)$ is transformed with matrix T.

The derivation of the new dependences between statements, however, has to take into account the fact that the computation spaces may have been applied different linear transformations. Consider statements S_w and S_r in the original code, where S_r is flow dependent on S_w. Let w be the write reference in S_w and r be the corresponding read reference. The flow dependence can be represented as:

$$write(w,\ d_{wr}\vec{I}) \rightarrow read(r,\ \vec{I})$$

which denotes that an array element read in iteration $\vec{I}$ due to the reference r to the array was written in iteration $d_{wr}\vec{I}$ due reference w to the array. If T_w is applied to $CS(S_w)$ and T_r is applied to $CS(S_r)$, then the dependence relation is then transformed to:

$$write(w,\ T_w d_{wr}\vec{I}) \rightarrow read(r,\ T_r\vec{I})$$

When the dependences continue to be uniform, then the following dependence will also exist, since the dependence relation is independent of particular values of $\vec{I}$:

$$write(w,\ (T_w d_{wr} - T_r)\vec{I}) \rightarrow read(r,\ \vec{0})$$

The dependence can be rewritten as:

$$write(w,\ d'_{wr}\vec{I}) \rightarrow read(r,\ \vec{0})$$

where $d'_{wr} = T_w d_{wr} - T_r$.

In general, the CDA transformation is legal iff all new dependence relations remain positive. This can be easily verified if the new dependences are uniform, such as when the matrices d_{wr}, T_w and T_r are of the form $\begin{bmatrix} I & u \\ 0 & 1 \end{bmatrix}$, where I is the identity matrix. In this case, we can verify that the write occurs earlier than the read by ensuring that the last column in d'_{wr} is lexicographically negative. If the dependences are non-affine, such as when the matrices d_{wr}, T_w and T_r are of the form $\begin{bmatrix} U & u \\ 0 & 1 \end{bmatrix}$, where U is any integer matrix, then more sophisticated techniques are necessary, such as those that reason with symbolic affine constraints [17, 46].

In our running example, the Computation Alignment changes the dependences from:

$$flow : \{(1,1),(0,1),(1,-1),(1,0)\}, anti : \{(0,1),(0,2)\}\}$$

to:

$$flow : \{(0,1),(1,0)\}, anti : \{(0,1)\}, output : \{(1,0)\}\}$$

There are cases, when after a CDA transformation the only violated dependences are loop independent flow dependences between statements. In this case, it is sometimes possible to make all dependences legal by textually interchanging some of the statements in the loop. Suppose, a statement S_r is flow dependent on statement S_w, and the new dependence is loop independent. The transformation applied is illegal if statement S_r appears before statement S_w in the text of the new loop body. However, the applied transformation can be made legal by textually interchanging the statements so that S_r is after S_w in the text of the new loop body.[6] In our running example,

[6] Textual interchange cannot make a transformation legal when statements S_w and S_r participate in a a cycle of loop independent flow dependences.

it is necessary to change the order of the statements in the loop so that $S_{1.1}$ is executed after $S_{1.2}$ and S_2 to maintain legality. Before the transformation, $S_{1.1}$ had a loop carried flow dependence from both $S_{1.2}$ and S_2. These dependences become loop independent after the alignment, thereby necessitating the reordering.

3.5 Generating New Loop Bounds

In this section, we discuss a technique to generate the bounds of a CDA transformed loop. The technique described here is a natural extension of the bound generation methods used in the linear loop transformation framework. The bounds of a linearly transformed loop are typically derived using either analytical techniques or the Fourier-Motzkin variable elimination technique [49], as described in Section 2.3.2. These techniques can also be used to derive the bounds of the transformed computation spaces. However, the derivation of the new loop bounds of the CDA transformed loop is more involved, since all transformed computation spaces together define the new iteration space.

The algorithm we describe here projects the transformed computation spaces onto an n-dimensional grid, which becomes the new iteration space. Algorithm CDA-bounds, outlined in Figure 3.7, directly adapts the Fourier-Motzkin elimination technique [3, 47, 49] to derive the bounds of a CDA transformed loop. Step 1 derives the new bounds for each of the computation spaces by variable elimination. We use $\beta(S_j)$ to denote the original bound matrix for computation space $CS(S_j)$, $1 \leq j \leq m$. If statement S_j is aligned by T_j, then The new bound matrix, $\beta'(S_j)$, can be computed from the original bound matrix as:

$$\beta'(S_j) = \beta(S_j)T_j^{-1}$$

The loop bounds of the transformed computation space are obtained from the new bound matrix by applying the Fourier-Motzkin variable elimination technique as described in Section 2.3.2 for linearly transformed loops.

Step 2 of CDA-bounds derives bounds for the new iteration space so that they subsume the bounds for all of the computation spaces. The new iteration space thus includes all the projected computation spaces. This is achieved by computing the lower bounds of the transformed iteration space as the minimum of lower bounds over all computations spaces. Similarly, the upper bounds are obtained by taking the maximum of upper bounds over all computation spaces. This step is similar to the derivation of the union of a set of data access descriptors [7].

Algorithm 1 : *CDA-bounds():*
/ Computes the bounds for CDA transformed loop */*
input: *Original loop bounds* L_i *and* U_i, $1 \le i \le n$ *and Computation Alignments*, $T_1, \ldots, T_m$
output: L'_i *and* U'_i *the new lower and upper bounds for iterators* $1 \le i \le n$

begin
  1.//Find new bounds for each computation space
    for $j = 1, m$
      $\mathcal{B}(S_j) \leftarrow S_j$ *bound matrix*
      $T_j \leftarrow$ *Transformation matrix for* $CS(S_j)$
      $\mathcal{B}'(S_j) \leftarrow \mathcal{B}(S_j)T_j^{-1}$
      for $i = n, 1$
        eliminate$(\mathcal{B}'(S_j), I_i)$
        // bounds for I_i *due to transformed* S_j
        $L'^j_i \leftarrow$ *max(lower bounds for* I_i*)*
        $U'^j_i \leftarrow$ *min(upper bounds for* I_i*)*
      end for
    end for
  2. // Derive subsuming bounds for the new iteration space
    for $i = 1, n$
      $L'_i \leftarrow min(L'^j_i, 1 \le j \le m)$
      $U'_i \leftarrow max(U'^j_i, 1 \le j \le m)$
    end for
end

Figure 3.7: Algorithm CDA-bounds to derive new loop bounds.

As an illustration of the algorithm, consider the bounds of the running example after the CDA transformation. Since the transformation matrices are simple offsets, it is easy to see that the new bounds of the computation spaces are:

$$CS(S_{1,1}): \quad 0 \le i \le n-1, \; 1 \le j \le n$$

$$CS(S_{1,2}): \quad 1 \le i \le n, \; 1 \le j \le n$$

$$CS(S_{1,1}): \quad 1 \le i \le n, \; 0 \le j \le n-1$$

The subsumption provides the following bounds for the new iteration space:

$$0 \le i \le n, 0 \le j \le n$$

```
for i = 0, n
  for j = 0, n
    S_{1.2}: (i > 0, j > 0)  A(i,j) = t(i,j) + B(i,j+1) + A(i,j-1)
    S_2:     (i > 0, j < n)  B(i,j) = A(i,j) + B(i,j+1)
    S_{1.1}: (i < n, j > 0)  t(i+1,j) = A(i,j) + A(i,j-1) + B(i,j)
  end for
end for
```

Figure 3.8: CDA transformed running example loop.

Algorithm CDA-bounds of Figure 3.7 is simple and retains the elegance of the bounds generation techniques used within the linear loop transformation framework. The new loop bounds are conservative in that the algorithm derives a convex polytope that subsumes the union of projected computation spaces, even though the union itself may not be convex. This does, however, have two problems. First, the iterations of the CDA transformed loop are not uniform in that not all iterations contain all of the computations. For instance, iteration (0, 1) of the transformed running example only executes statement $S_{1.1}$, but not $S_{1.2}$ and S_2. Second, the transformed loop may have new, previously non-existing iterations with no computations to execute. For instance, iteration (0, 0) of the CDA transformed running example does not correspond to any iteration of the original loop, and its execution is unnecessary. For these reasons, a mechanism is needed that enables the computations in the transformed loop to be executed that should be executed, but disables those that should not. We use for this purpose conditional statements called *guards* in the loop body that allow the execution of appropriate computations and prevent the execution of the other computations.

Definition 8 (Guard) *A guard $g(S)$ for a statement S is a conditional statement whose condition is true only in those iterations that are within the bounds of the transformed computation space of S.* □

The guard for a statement is just a conjunction of conditions on the iterators. Figure 3.8 shows the guards inserted in the loop of the running example. The guard for $S_{1.2}$ prevents its execution in iteration (0, 0). Similarly, the guard for S_2 prevents its execution in iterations $(0, *)$ and $(*, n)$, and the guard for $S_{1.1}$ prevents its execution in iterations $(n, *)$ and $(*, 0)$, where $*$ denotes all integers i such that $0 \leq i \leq n$.

The derivation of the loop bounds for the transformed loop of the running example was relatively simple because the transformation matrices were simple offsets. In order to illustrate the derivation of loop bounds with general integer transformation matrices, consider the loop of Figure 3.9. The

```
for i = 0, n
  for j = 0, n
    S1 :  A(i, j) = A(i - 1, j - 1)
    S2 :  B(i, j) = A(i + j, j)
  end for
end for
```

Figure 3.9: An example loop used to illustrate overhead of empty iterations.

bounds of both $CS(S_1)$ and $CS(S_2)$ in the original loop are:

$$0 \le i \le n, 0 \le j \le n$$

which can be represented by bound matrices $\beta(S_1)$ and $\beta(S_2)$:

$$\beta(S_1) = \beta(S_2) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & 0 & n \\ 0 & -1 & n \end{bmatrix}$$

Suppose $CS(S_1)$ is applied the identity transformation and $CS(S_2)$ is applied transformation T_2:

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

In this case:

$$T_2^{-1} = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The bound matrix for $CS(S_1)$ does not change, but the new bound matrix $\beta'(S_2)$ becomes:

$$\beta'(S_2) = \beta(S_2)T_2{-}1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & 0 & n \\ 0 & -1 & n \end{bmatrix} \begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & 0 \\ -1 & 1 & n \\ 0 & -1 & n \end{bmatrix}$$

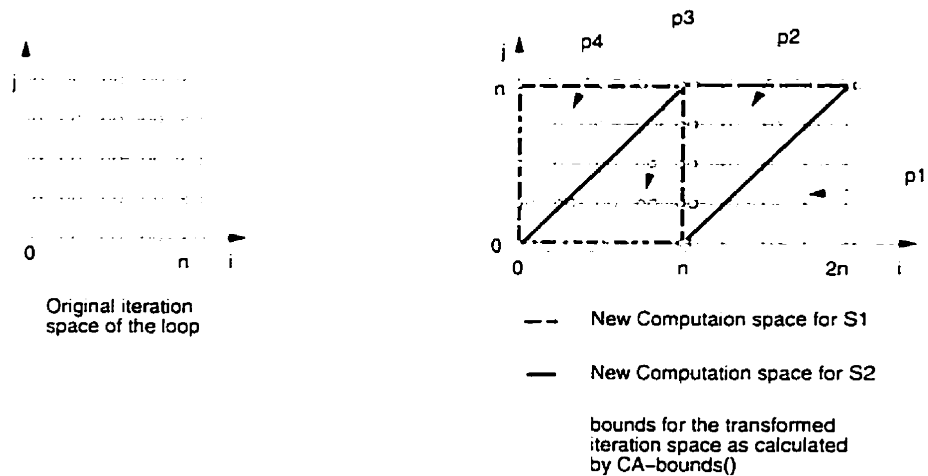


Figure 3.10: Deriving new loop bounds.

Applying Fourier-Motzkin variable elimination to $\mathcal{J}'(S_2)$ results in the bounds:

$$0 \leq i \leq 2n \quad and \quad max(0, i-n) \leq j \leq min(n, i)$$

Combining the bounds for the two transformed computation spaces, we obtain the following bounds for the transformed loop:

$$0 \leq i \leq 2n \quad and \quad min(0, max(0, i-n)) \leq j \leq max(n, min(n, i))$$

which can be simplified to be:

$$0 \leq i \leq 2n \quad and \quad 0 \leq j \leq n$$

These bounds subsume the bounds of the individual transformed computation spaces $CS(S_1)$ and $CS(S_2)$.

Figure 3.10 shows the original iteration space on the left hand side and the transformed computation spaces on the right hand side. The conservative bounds, $0 \leq i \leq 2n$ and $0 \leq j \leq n$, of the transformed iteration space, as calculated by CDA-bounds, are shown as a dashed box. The transformed iteration space consists of four regions marked $p1$ to $p4$. Iterations in $p1$ have neither of the two computations; iterations in $p2$ have only S_2 computations; iterations in $p3$ have both S_1 and S_2 computations; and finally iterations in $p4$ have only S_1 computations. Clearly, statements S_1 and S_2 must have guards to ensure correct execution of the transformed loop. The guards for S_1 and S_2 in the transformed loop evaluate conditions $0 \leq i \leq n \ \wedge \ 0 \leq j \leq n$ and

```
for  i = 0, 2 * n
  for  j = max(0, i - n), n
    S1 :  (i ≤ n)                            A(i,j) = A(i - 1, j - 1)
    S2 :  (j ≥ i - n,  j ≤ min(i, n))        B(i - j, j) = A(i, j)
  end for
end for
```

Figure 3.11: Transformed loop after simple guard optimizations.

$0 \leq i \leq 2n \ \wedge \ max(0, i-n) \leq j \leq min(n, i)$, respectively.

Generating the loop bounds this way is sufficient when the majority of the new iterations contain all the statements. The running example loop was transformed with matrices that have small offsets so there were only two empty iterations. However, the transformed iteration space of Figure 3.10 has as many as a quarter of the iterations that contain no computations. It is possible to optimize the bounds to minimize the number of empty iterations. For instance, modifying the lower bound of j from $min(0, max(0, i-n))$ to $max(0, i-n)$, steps off all empty iterations in region $p1$ of the new iteration space. This type of optimization is important because the evaluation of guards can add significant run-time overhead due to the additional computation for checking the inequalities. In the next chapter, we discuss techniques to derive CDA transformed loops with minimal empty iterations, and we discuss techniques to minimize the overhead of guard computations.

3.6 Applications of CDA

CDA transformations can be used to optimize nested loops in a number of contexts. For example, CDA can be used to perform the following optimizations:

- reducing the number of cache conflicts,
- improving the efficiency of parallel SPMD (Single Program Multiple Data) programs,
- improving instruction level parallelism,
- eliminating barrier synchronizations,
- improving loop performance using CDA as generalized loop distribution transformation, and
- improving loop performance by transforming certain imperfect loop nests.

The first two optimizations are covered in Chapters 5 and 6 respectively. The remaining optimizations are covered in Chapter 7.

3.7 Disadvantages of CDA

Additional optimization opportunities that CDA provides are at an additional cost because CDA transformations also have some disadvantages:

- Good heuristics are the key to efficient derivation of CDA transformations. The relatively fine-grained restructuring that is possible within the CDA framework implies a vastly larger search space than when deriving a linear loop transformation. In the following chapters we show that, with the knowledge of the optimization context, CDA transformations can be derived efficiently.
- CDA transformed loops typically have more overheads than linearly transformed loops in that they have empty iterations and guard computations and require storage for temporary variables. In the next chapter, we describe methods that substantially reduce these overheads.

Chapter 4

Optimizing CDA Transformed Loops

Any change or reform you make is going to have consequences you don't like.
— *Udall's Fourth Law*

The previous chapter presented the basic CDA transformation technique and the formalism underlying the CDA framework. It was noted that CDA transformations introduce overheads that purely linear transformations would not. These overheads include empty iterations, guard computations and additional storage for temporary variables.

In this chapter, we focus on techniques to improve the efficiency of CDA transformed loops by reducing these overheads. The techniques we present can be quite effective. We illustrate the effect of the techniques on the overhead generated by two different the CDA transformations on the loop of Figure 4.1. The transformed iteration space for the first transformation is shown on the left hand side of Figure 4.2, where one of the computation spaces is applied an offset alignment of (k, k), where k is a positive integer. The left hand side of Figure 4.3 shows the transformed iteration space for the second transformation, where one of the computation spaces is skewed with respect to the other. We will refer to the CDA transformed loops for these two transformed iteration spaces as *Loop 1* and *Loop 2*, respectively. For the purpose of the experiments the loop size n was set to 1000 and k was set to 5, unless otherwise specified. Also, the execution time measurements were taken on a SUN workstation with hyperSPARC CPU.

The overhead of *Loop 1* with the loop bounds generated by algorithm *CDA-bounds* of Section 3.5 is shown as first five bars on the right hand side of Figure 4.2. This overhead is mainly due to empty iterations and guard computations; it increases slightly with an increasing k, due to increasing number of empty iterations and guard computations. For $k = 5$, this overhead is about 22% of the execution time of the original loop. The overhead can be reduced significantly by optimizing the CDA transformed loop with techniques described in this chapter. The last five bars on the right hand side of Figure 4.2 correspond to the optimized *Loop 1*, where the overhead is less than 0.1%

```
for i = 0,n
  for j = 0,n
      U(i,j) = c(0) * U(i,j)
      R(i,j) = c(0) * R(i,j)
  end for
end for
```

Figure 4.1: The loop used to illustrate the effect of techniques to reduce overheads.

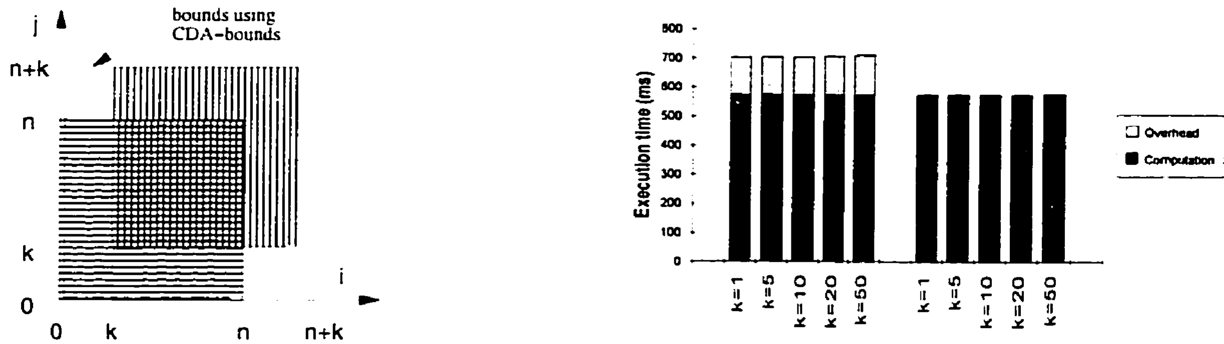


Figure 4.2: Overheads in a CDA transformed loop, called *Loop 1*, with offset alignment (k, k). In the bar chart above, the bars on the left correspond to the execution times of *Loop 1* with overheads, whereas the bars on the right correspond to the execution times of *Loop 1* after reducing overheads with techniques described in this chapter.

of the original loop. The optimized loop has neither empty iterations nor guard computations.

The overheads can also be reduced significantly when the alignments are more general than offsets. The overhead of *Loop 2* with the bounds generated by algorithm *CDA-bounds* of Section 3.5 is shown as the first bar on the right hand side of Figure 4.3. The overhead can be much higher than when using offset alignments (nearly 78% of the original loop in this case), since nearly one quarter of the iterations are empty. However, the overhead is reduced to about 5% of the original loop when *Loop 2* is optimized by removing empty iterations and guards, using the techniques described here.

In this chapter, we describe:

- an algorithm to tighten the bounds so as to reduce the execution overhead of empty iterations in the transformed loop;
- methods to reduce run-time overhead of guard computations; and finally
- techniques to reduce the storage requirement for temporary variables.

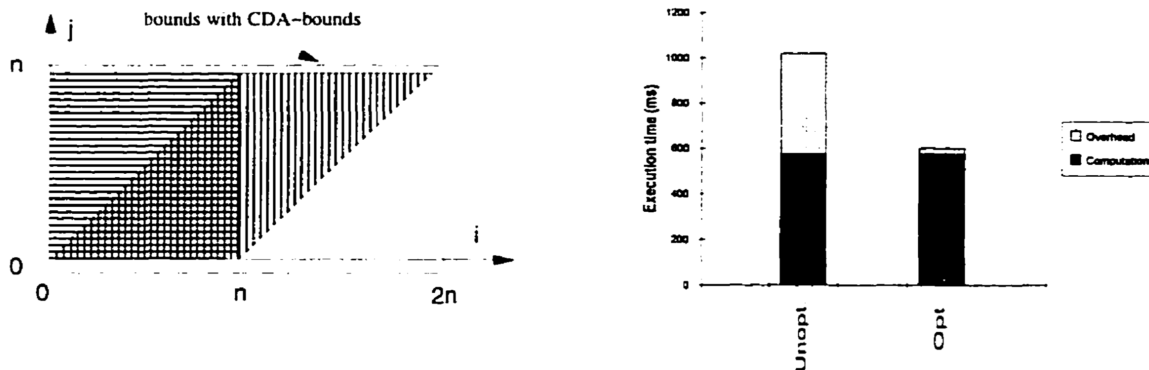


Figure 4.3: Overheads in a CDA transformed loop with a linear alignment, called *Loop 2*.

4.1 Removing Empty Iterations

The iteration space of a CDA transformed loop is the union of the transformed computation spaces projected onto an integer space (which we refer to as the union of computation spaces for conciseness). However, the bounds of the CDA transformed loop, as derived in Section 3.5, were chosen so that the new loop scans integer points in a convex polytope which is a superset of the union of computation spaces. Therefore, the derived loop bounds will not necessarily be tight in that the CDA transformed loop will contain empty iterations (namely, those outside the union of computation spaces), which do not contain the execution of any of the statements. In this section, we show how to derive tight loop bounds so that a CDA transformed loop scans integer points in the smallest convex polytope containing the union of computation spaces. With tighter loop bounds, the overhead of empty iterations and the guard computations they contain is reduced.

While deriving tight loop bounds, it is desirable to keep the CDA transformed loop perfectly nested, because it may be necessary to apply other loop transformations later on, and most transformations require that the loop be perfectly nested. In order to obtain a perfectly nested CDA transformed loop, the polytope that the loop scans must be convex; only in some cases do non-convex polytopes correspond to perfect nestings. Some examples are shown in Figure 4.4. The convex polytope of Figure 4.4(a) corresponds to a perfectly nested loop with simple integral, affine expressions of iterators in the loop bounds. Some non-convex polytopes, as in Figures 4.4(b) and (c), also correspond to perfectly nested loops, where the loop bound expressions contain minimum or maximum operators on integral, affine functions of iterators. These two non-convex polytopes happen to have the property that for given range of I_1 values, there exists a continuous range of

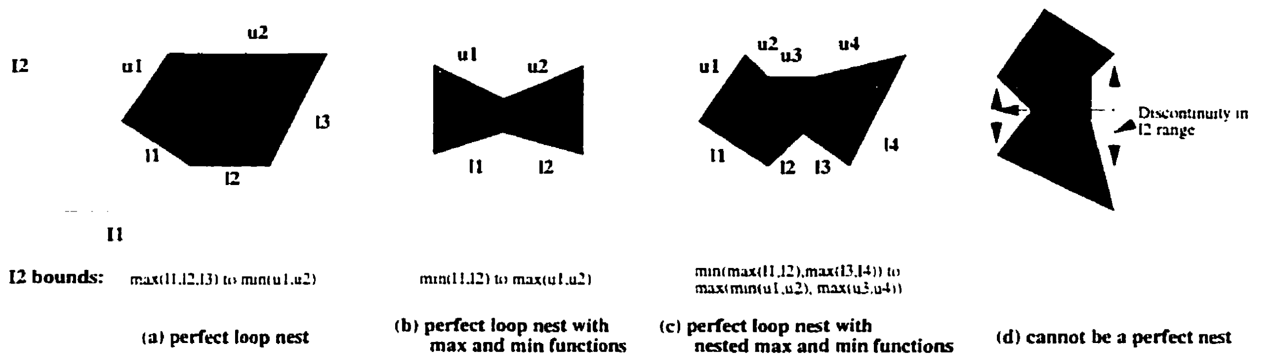


Figure 4.4: Shapes of iteration spaces

I_2 values, which defines the loop iterations.[1] On the other hand, the non-convex polytope shown in Figure 4.4(d) cannot be represented by a perfectly nested loop, since for certain ranges of I_1 values, the range of I_2 values is discontinuous, requiring a separate I_2 loop inside the I_1 loop for each continuous I_2 range.

The technique described in this section derives tight loop bounds by constructing the convex-hull [44] of the union of computation spaces. Algorithm *CDA-bounds-perfect* of Figure 4.5 finds this convex-hull.[2] The algorithm finds the smallest convex polytope that contains the union of the computation spaces. When the union is a convex polytope itself, then the derived loop bounds are exact in that the transformed loop does not have any empty iterations.

Step 1 of algorithm *CDA-bounds-perfect* finds the extreme points, E_i, of each transformed computation space, $CS'(S_i)$. The extreme points of a polytope are integer points on the boundaries of the polytope so that the convex-hull of these points bounds the polytope. For example, the four corners of a rectangular area are its extreme points, since a convex-hull of the corners defines the rectangle.

The extreme points of a computation space are obtained from its bound matrix as follows. The inequalities represented by the bound matrix of a computation space define half-spaces, and the intersection of all half-spaces defines the integer points in the computation space. The equations for the hyperplanes bounding a computation space can therefore be obtained by replacing $\geq$ and $\leq$ operators in the inequalities by the $=$ operator. The extreme points of an n-dimensional computation space are obtained by solving combinations of n hyperplane equations. The solutions for these combinations of equations are points where the hyperplanes intersect. Out of these solutions, the

[1] There may be periodical "holes" left by non-unit strides due to non-unimodular transformations.

[2] The word *perfect* in the name of the algorithm highlights that the objective of the algorithm is to generate a perfectly nested transformed loop.

Algorithm 2 : *CDA-bounds-perfect*
/ Computes tight bounds. */*
input: *bounds for computation spaces* $CS'(S_1), \ldots, CS'(S_m)$
output: *bound matrix* β' *of the smallest convex polytope containing the computation spaces*
begin
1. *for* $i = 1, m$
   $E_i \leftarrow$ *extreme points of* $CS'(S_i)$
   end for
2. $H \leftarrow Convex_hull(\cup_{i=1,m} E_i)$
3. $u \leftarrow$ *any point such that* $u \in CS'(S_i)$, *for some* i, $1 \leq i \leq m$
4. $\beta \leftarrow \phi$
5. *for each bounding hyperplane* $(h(\vec{I}) = 0) \in H$
   if $h(u) \leq 0$ *then*
   $\beta \leftarrow \beta \cup (h(\vec{I}) \leq 0)$
   else
   $\beta \leftarrow \beta \cup (h(\vec{I}) \geq 0)$
   end if
   end for
6. $\beta' \leftarrow Fourier_Motzkin(\beta)$

end

Figure 4.5: Tight transformed bounds

---

ones that are integer points within the computation space are chosen as desired extreme points, $E_1, \ldots, E_m$.

Step 2 of the algorithm computes the convex-hull of the union of extreme points, $E_1 \cup \ldots \cup E_m$, by applying any of the well known techniques such as the *gift-wrapping* or the *beneath-beyond* methods [44]. The convex-hull is defined by a set, H, of bounding hyperplanes of the form $h(\vec{I}) = 0$, each being either a lower bound or an upper bound of the transformed iteration space. Whether it is an upper bound or a lower bound depends on which side of the hyperplane an integer point in the union of computation spaces lies.

For this purpose, Step 3 chooses an arbitrary point u known to be in the union of computation spaces. In Step 4, a set of inequalities, β, is initialized to the empty set. Step 5 adds an inequality to β in each iteration so that at the end of the iterations, β represents the bounds of the smallest convex polytope containing the union of the computation spaces. Whether a hyperplane is added as an upper or a lower bounding hyperplane is determined using integer point u: For hyperplane $h(\vec{I}) = 0$ in H, $h(\vec{I}) \leq 0$ is added to β if $h(u) \leq 0$; otherwise $h(\vec{I}) \geq 0$ is added to β. Finally, in Step 6, we apply Fourier-Motzkin variable elimination to β to obtain the bound matrix β'.

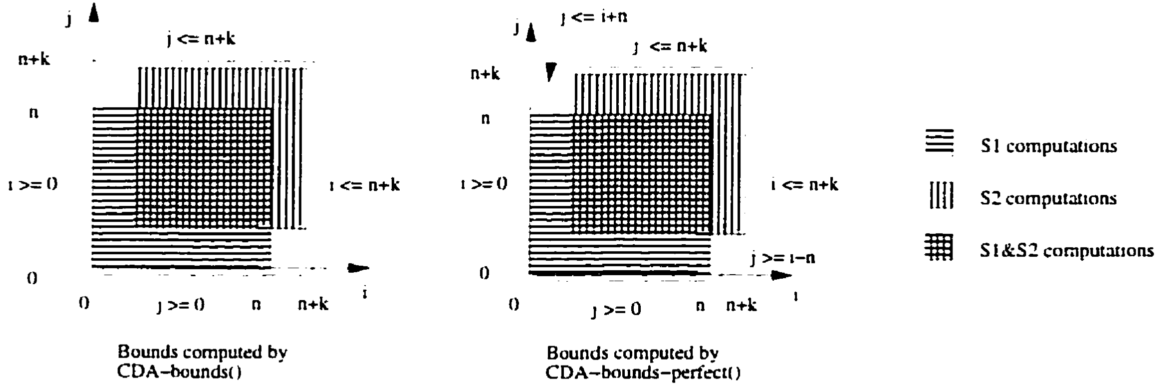


Figure 4.6: Empty iterations in an iteration space with tight bounds.

As an example of applying algorithm *CDA-bounds-perfect*, consider again the transformed computation spaces for *Loop 2* on the left hand side of Figure 4.3. The extreme points of the first transformed computation space, $CS'(S_1)$, are $(0, 0)$, $(n, 0)$, $(0, n)$ and (n, n), and the extreme points for the second transformed computation space, $CS'(S_2)$, are $(0, 0)$, $(n, 0)$, $(2n, n)$ and (n, n). Therefore, Step 2 in the algorithm computes the convex-hull as defined by the lines[3]:

$$i = 0, \quad i = 2n, \quad j = 0, \quad j = n, \quad j = i - n$$

from which Step 5 produces the following inequalities:

$$i \geq 0, \quad i \leq 2n, \quad j \geq 0, \quad j \leq n, \quad j \geq i - n$$

After variable elimination, these inequalities provide the loop bounds,

$$0 \leq i \leq 2n, \qquad max(0, i - n) \leq j \leq n$$

These inequalities bound the shaded area in the figure. The loop bounds are exact in this case, since the union is a convex polygon, so it no longer includes empty iterations.

In some cases, the bounds derived using algorithm *CDA-bounds-perfect* are in-exact in that not all empty iterations are removed. Consider the union of the computation spaces of *Loop 1* depicted on the left hand side of Figure 4.2, where the union is a non-convex polygon. The dotted lines on the left hand side of Figure 4.6 show the loop bounds that are derived by algorithm *CDA-bounds* of Section 3.5. The dotted lines at the center of the figure show the bounds obtained by using

[3] Because we are operating in a 2-dimensional space, the hyperplanes are actually lines.

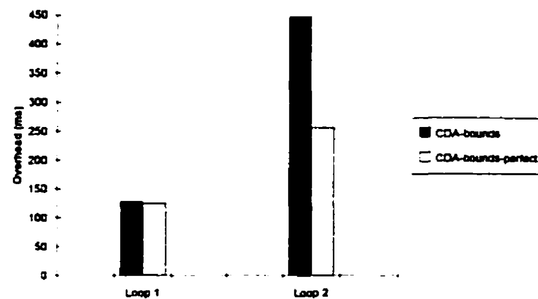


Figure 4.7: Performance benefits of eliminating empty iterations.

algorithm *CDA-bounds-perfect*.[4]

Figure 4.7 compares the overhead of the unoptimized *Loops 1* and *2*, where the bounds are derived using algorithm *CDA-bounds*, with the overhead of the optimized loops, where the bounds are derived using algorithm *CDA-bounds-perfect*. The reduction in the overhead of *Loop 1* (of Figure 4.2) is not significant, since it contains only a small number of empty iterations. The application of algorithm *CDA-bounds-perfect* to *Loop 2* (of Figure 4.3) reduces the overhead by about 45%, since nearly one quarter of its iterations were empty.

4.2 Reducing the Overhead of Guard Computations

Guards are often necessary in CDA transformed loops both to step off empty iterations and to prevent inappropriate computations from executing in the new iterations. Guards may incur considerable run-time overhead, but it is possible to remove them during compilation time in many cases. We describe three techniques to reduce the number of guard computations required:

i) Algorithm *CDA-bounds-perfect* described in the previous section minimizes the number of empty iterations and hence removes the guard computations in the eliminated empty iterations (although it does not eliminate all the guard computations). The algorithm can be applied in conjunction with and prior to applying techniques (*ii*) and (*iii*) below.

[4]Algorithm *CDA-bounds-perfect* can be used in contexts other than CDA transformations, for example when computing the union of *Data Access Descriptors* [7]. Data access descriptors are descriptions of sections of arrays accessed in a loop or a procedure. The union of data access descriptors might be computed, for instance, to determine whether two procedures access the same set of array elements. The data access descriptors can be represented by bound matrices, which allows algorithm *CDA-bounds-perfect* to be used to compute the union of data descriptors. Algorithm *CDA-bounds-perfect* can also be useful in other contexts that require the computation of unions such as array privatization [53, 16]. In both of these contexts, the algorithm provides tighter unions than the existing algorithms prevalently do.

ii) The second technique incrementally eliminates guard computations from the iterations in those regions of the new iteration space in which all statements are to be executed. This technique is useful, for example, when most iterations must execute all statements. (This typically occurs when the transformation matrices for the statements are similar.)

iii) In the third technique, the union of the computation spaces is partitioned into *homogeneous segments*, where each segment must execute the same set of statements. The technique then generates a loop structure that iterates through the segments, where the loop body of the subnest corresponding to each segment contains only those statements that the segment must execute. Although this technique can eliminate most of the guard computations, it usually generates complex and imperfect transformed loops. Therefore, this technique would typically be used at a final stage to eliminate guard computations that might remain after any other loop transformations that may be applied and after the application of technique (*i*).

We describe techniques (*ii*) and (*iii*) above in the following two subsections.

4.2.1 Incremental Removal of Guards

This section describes a technique that incrementally removes guards from selected regions of the union of computation spaces. It is targeted primarily towards CDA transformations, where the intersection of the computation spaces makes up a large portion of the union of the computation spaces.

The technique we describe here is somewhat involved, so we first describe it with an example, namely the transformed iteration space on the left of Figure 4.8.[5] We refer to the CDA transformed loop corresponding to this iteration space as *Loop 3*. Algorithm *CDA-guard-rem* of Figure 4.10 removes guards using the following steps:

1. The bounds of the intersection of the computation spaces are derived. For instance, the shaded area in Figure 4.8 is the intersection of the three computation spaces. The iterations in the intersection entail the execution of all three statements S_1, S_2 and S_3. Therefore, if we partition the new iteration space to separate out the intersection, the code generated for the iterations in the intersection does not require any guards.[6]

[5] The CDA transformations in the figure are such that the computation space of statement S_2 is moved up by k in the I_2 direction with respect to the computation space of statement S_1, and the computation space of statement S_3 is moved right by k in the I_1 direction with respect to the computation space of statement S_1.

[6] Thus, algorithm *CDA-guard-rem* generates code so that all iterations of the intersection are in a subnest of their own.

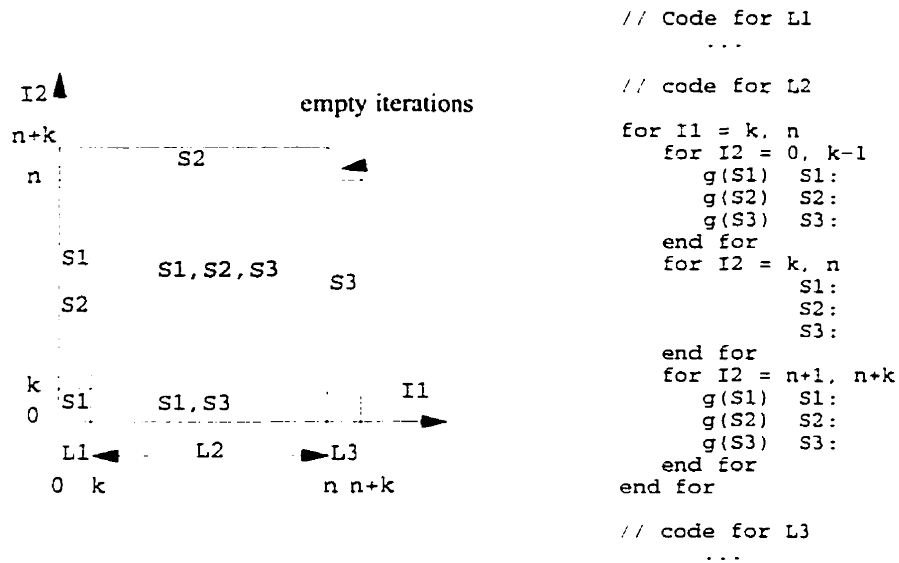


Figure 4.8: Transformed computations spaces to illustrate steps in algorithm *CDA-guard-rem*. The transformed loop corresponding to the transformed computation spaces is called *Loop 3*.

2. The iteration space is partitioned along the first dimension I_1 so as to delineate the intersection in that dimension. In our example, the CDA transformed iteration space of Figure 4.8 is divided into three partitions, namely, L_1, L_2 and L_3, based on the fact that the I_1 bounds for the intersection are k and n. Partition L_1 has iterations with I_1 values between 0 and $k-1$; partition L_2 has iterations with I_1 values between k and n, (the two I_1 bounds for the intersection); and partition L_3 has iterations with the I_1 values between $n+1$ and $n+k$.

3. Code is generated for partition L_2. This code consists of a sequence of subnests. The first subnest includes those iterations with I_2 values that do not belong to the intersection, thus requiring guards. The second subnest includes the iterations that belong to the intersection. This code constitutes most of the iterations of the loop that need to be executed and require no guards. The final subnest includes those iterations with I_2 values higher than those of the intersection, thus requiring guards again. The three subnests for our example are shown on the right hand side of Figure 4.8. Note that the subnest corresponding to the intersection does not have any guard computations.

4. The algorithm is applied recursively to remove guards from partitions L_1 and L_3. The iterations in these partitions contain only a subset of the statements of the original loop body. Thus, only a subset of the computation spaces participate in the intersections of these partitions. For partition L_1, it is necessary to consider only the computation spaces for statements

Algorithm 3 : *CS-intersect(L)*
input: *Loop L with loop body* $g(S_1) : S_1; \ldots g(S_K) : S_K$
output: *Bounds of the intersection of the computation spaces*
begin
  // $\beta(S_1), \ldots, \beta(S_K) \leftarrow$ *bounds for transformed* $CS(S_1), \ldots, CS(S_K)$
  // $\beta(L) \leftarrow$ *bounds of iteration space partition*
  1. for each S_i, $1 \leq i \leq K$
    if Fourier_Motzkin($\beta(S_i) \cup \beta(L)$) *inconsistent then*
      remove S_i *from L*
    end if
  end for
  2. Remove L if all statements are removed
  3. for $i = n, 1$
    // *Bounds of iterator* I_i *in* $CS(S_j)$ *are* $L_i^j \leq I_i \leq U_i^j$.
    $L'_i \leftarrow max(L_i^j, 1 \leq j \leq K)$
    $U'_i \leftarrow min(U_i^j, 1 \leq j \leq K)$
  end for
  4. $\beta(CI) \leftarrow$ *bound matrix for the intersection*
end

Figure 4.9: Merging Guards for statements $S_1, \ldots, S_K$.

S_1 and S_2, and for partition L_3 it is necessary to consider only computation spaces for statement S_3. Recursive application of the algorithm to partition L_1, does not partition it further along I_1, since the intersection of computation spaces for S_1 and S_2 spans the entire I_1 bounds of L_1. The intersection in L_1 has I_2 bounds of k and n, and guards can be similarly removed from L_1.

The result of applying algorithm *CDA-guard-rem* is thus a sequence of loop nests, which typically are imperfectly nested. The right hand side of Figure 4.8 shows a template of the code generated for the transformed computation spaces on the left hand side. The technique in *CDA-guard-rem* is similar to the method by Knijnenburg and Bik [24], where a perfectly nested loop with a single `if (condition) then-else-endif` statement in the loop body is restructured so as to eliminate the need for evaluating the integral, affine `condition` in the statement.

In order to simplify the description of algorithm *CDA-guard-rem*, a loop L and its iteration space are used interchangeably, and the notation for its bound matrix and that for the set of inequalities it represents are used interchangeably. The input to the first invocation of algorithm *CDA-guard-rem* is the entire iteration space of the CDA transformed loop; the input to later, recursive invocations is

---

Algorithm 4 : *CDA-guard-rem(L)*
input: *Loop L guarded with* $g(S_1) : S_1; \ldots g(S_K) : S_K$
output: *Code for L with fewer guards*
begin
  // $\beta(L) \leftarrow$ *bounds for L*
  1. $\beta(CI) \leftarrow$ *CS-intersect(L)*
    // *CS-intersect returns* $\beta(CI)$, *bounds of intersection of* $S_1, \ldots, S_K$
  2. $L_1^0 \leq I_1 \leq U_1^0 \leftarrow I_1$ *bounds in* $\beta(L)$
  3. $L_1 \leq I_1 \leq U_1 \leftarrow I_1$ *bounds in* $\beta(CI)$
  4. Generate three partitions, L_1, L_2 *and* L_3
      $\beta(L_1) \leftarrow \beta(L) \cup \{L_1^0 \leq I_1 \leq L_1 - 1\}$
      $\beta(L_2) \leftarrow \beta(L) \cup \{L_1 \leq I_1 \leq U_1\}$
      $\beta(L_3) \leftarrow \beta(L) \cup \{U_1 + 1 \leq I_1 \leq U_1^0\}$
  // *Remove guards in* L_1, L_2 *and* L_3
  5. CDA-guard-rem(L_1*)*
  // *generate code for subnest* L_2
  6. print: for $I_1 = L_1, U_1$
      if $n > 1$ *then // the loop is at least 2-dimensional*
          Gen-subnest($\beta(L), \beta(CI)$, *2)*
      end if
    print: end for
  7. CDA-guard-rem(L_3*)*
end

Figure 4.10: Remove guard computations in L.

---

a portion of the iteration space. We assume that the bounds of the iteration space were generated by algorithm *CDA-bounds-perfect* and that the bounds of the iteration space partitions were generated during the previous invocations of algorithm *CDA-guard-rem*.

The first step of algorithm *CDA-guard-rem* invokes algorithm *CS-intersect*, which primarily generates the bound matrix $\beta(CI)$ for the intersection of the computation spaces. (The 'CI' in $\beta(CI)$ stands for Computation space Intersection.)

In algorithm *CS-intersect* of Figure 4.9, L corresponds to the iteration space under consideration, and $\beta(L)$ denotes its bounds. Step 1 identifies those statements that are not executed in any iteration of L (a case, that only occurs on recursive invocations of *CDA-guard-rem*). Such statements are recognized by checking whether their guards are inconsistent with $\beta(L)$. If the guards are indeed inconsistent with $\beta(L)$, then the guards will not be true in any iteration of L. Hence, such statements can be removed from L. For instance, a guard $i < n$ is inconsistent with $\beta(L)$, when $\beta(L)$ contains the inequality $i > n$. If none of the guards are consistent with $\beta(L)$, then L

is a partition containing only empty iterations, and can consequently be removed (Step 2). Step 3 computes the bounds for the intersection of the computation spaces of the statements. These bounds are represented in bound matrix form in Step 4.

We now continue with the description of algorithm *CDA-guard-rem*, shown in Figure 4.10. Steps 2 and 3 extract bounds along I_1 for iteration space partition L and the intersection CI of the computation spaces, respectively. These I_1 bounds are used in Step 4 to derive the bounds for partitions L_1, L_2 and L_3 of partition L. Partition L_1 consists of iterations in L for which I_1 values lie between the lower bound of I_1 in L and the lower bound of I_1 in CI. Partition L_2 has iterations in L for which I_1 values lie within the I_1 bounds of CI. Partition L_3 has the remaining iterations: that is, those iterations in L for which the I_1 value lie between the upper bound of I_1 in CI and the upper bound of I_1 in L. Such a partitioning of the iteration space is always possible, because CI is contained entirely within L.[7] We then recursively apply algorithm *CDA-guard-rem* to partitions L_1 and L_3, in Steps 5 and 7, respectively. This recursive application removes guards by isolating the intersection of (fewer) computation spaces from L_1 and L_3, respectively.

In Step 6, the subnest corresponding to the iterations in partition L_2 is generated. This involves generating a loop statement that iterates from the lower bound of CI to the upper bound of CI; and contains a sequence of subnests, as generated by algorithm *Gen-subnest*.

Algorithm *Gen-subnest*, shown in Figure 4.11, generates code for each partition created by algorithm *CDA-guard-rem*. *Gen-subnet* is recursive, and each invocation produces the loop statements necessary for the next iterator I_r. The iteration space is partitioned along the rth dimension by identifying three ranges for iterator I_r, similar to the partitioning of the iteration space along the first dimension in algorithm *CDA-guard-rem*. Steps 1 and 2 extract the bounds for iterator I_r in L and CI, respectively. In Step 3, we print a loop statement for iterator I_r that iterates from the lower bound of I_r in L to one less than the lower bound of I_r in CI. The code for the remaining iterators I_{r+1} to I_n of this subnest (assuming the loop has dimension n) is generated using their bounds in L, since these iterations are not in CI. In Step 4, we then generate a loop statement for iterator I_r using the bounds for I_r in CI. The code for the remaining iterators is generated by a recursive call to algorithm *Gen-subnest*. Finally, Step 5 generates a loop statement for iterator I_r that iterates from the upper bound of I_r in CI plus one to the upper bound of I_r in L. The code for the remaining iterators I_{r+1} to I_n of this subnest is generated as in Step 3.

[7]That is, for each iterator, the lower bounds of CI are greater than or equal to the lower bounds of L and the upper bounds of CI are lesser than or equal to the upper bounds of L.

Algorithm 5 : *Gen-subnest*($\mathcal{I}(L)$, $\mathcal{I}(CI)$, r)
input: *Loop L, intersection CI and nesting level r*
output: *A sequence of subnest without guards for the intersection*
begin
 1. $L_r^0 \leq I_r \leq U_r^0 \leftarrow$ I_r *bounds in* $\mathcal{I}(L)$
 2. $L_r \leq I_r \leq U_r \leftarrow$ I_r *bounds in* $\mathcal{I}(CI)$
 // *This subnest has guards*
 3. *print:* for $I_r = L_r^0, L_r - 1$
  if $r < n$ *then*
   Generate code for iterators I_{r+1} *to* I_n *using existing bounds in L*
  else return
  end if
  print: end for
 // *Only some subnests here have guards*
 4. *print:* for $I_r = L_r, U_r$
  if $r < n$ *then*
   Gen-subnest($\mathcal{I}(L)$, $\mathcal{I}(CI)$, $r + 1$)
  else return
  end if
  print: end for
 // *This subnest has guards*
 5. *print:* for $I_r = U_r + 1, U_r^0$
  if $r < n$ *then*
   Generate code for iterators I_{r+1} *to* I_n *using existing bounds in L*
  else return
  end if
  print: end for
end

Figure 4.11: Generation of code for subnests.

Figure 4.12 shows the effectiveness of algorithm *CDA-guard-rem* in removing guards. The dark bars correspond to CDA transformed code with guards, where algorithm *CDA-bounds-perfect* was applied to remove as many empty iterations as possible. The grey bars correspond to code for which algorithm *CDA-guard-rem* was applied. The figure shows that additional removal of guards can reduce the overhead substantially, when the loops are transformed by offset alignments. The reduction in overhead for *Loop 2* was not as large as for *Loops 1* and *3*, since the code for *Loop 2* continues to have guards in nearly one quarter of the iterations, but the benefits of applying *CDA-guard-rem* is still significant.[8]

[8] These iterations correspond to the region bounded by $0 \leq i \leq n$ and $i + 1 \leq j \leq n$ on the left of Figure 4.3.

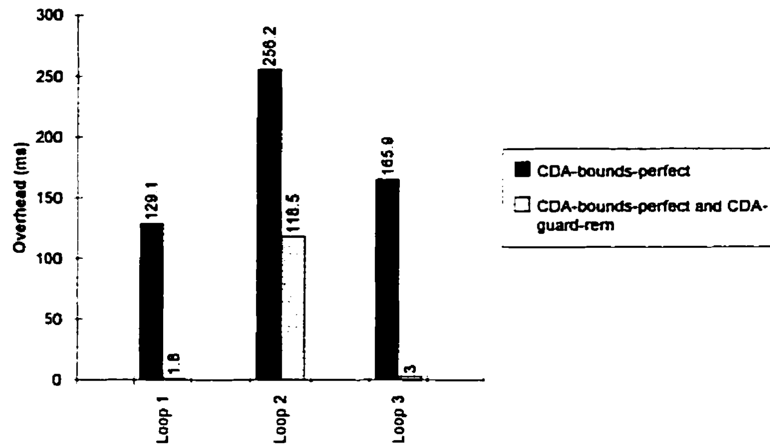


Figure 4.12: Performance benefits of removing guards by *CDA-guard-rem*.

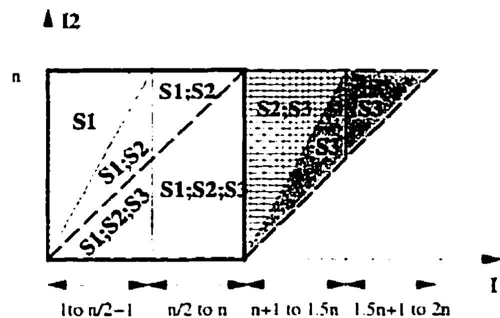


Figure 4.13: Partitioning union into homogeneous segments.

4.2.2 Partitioning the Iteration Space into Homogeneous Segments

The technique for incremental elimination of guards can be further refined so that the transformed iteration space is partitioned into homogeneous *segments*, where all iterations in a segment execute exactly the same set (although not necessarily all of the statements). The code generated would then iterate separately through each of these segments, and the subnest generated for each segment would contain only those statements in the segment that need to be executed; no guards would be necessary. We refer to this technique as *Homogeneous-partitioning*. The idea behind *Homogeneous-partitioning* is utilized in several existing algorithms [12, 22, 23, 24, 27, 51, 52].

In this section, we only illustrate the basic approach in *Homogeneous-partitioning* by example through the iteration space of Figure 4.13. The iteration space is partitioned into eight homogeneous segments, and loop dimensions I_1 and I_2 are partitioned so as to demarcate these segments.[9] Given these segments, *Homogeneous-partitioning* generates a sequence of four nested loops, one for each

[9] Actually, the union has only five homogeneous segments, but they are easier to work with when treated as eight segments.

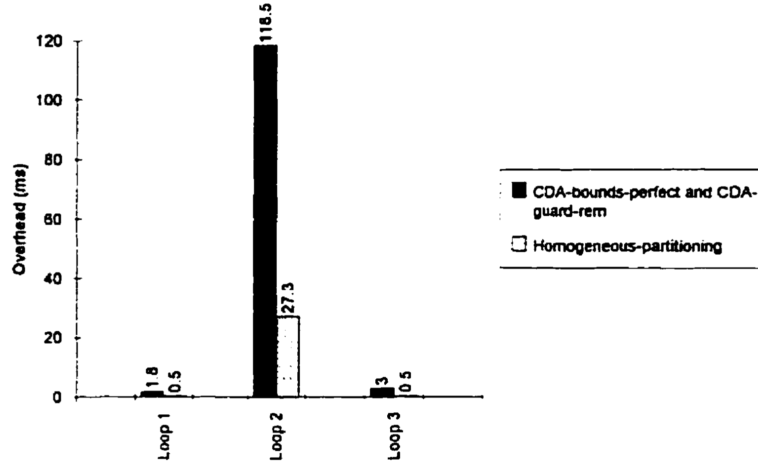


Figure 4.14: Performance benefits of scanning in homogeneous segments.

defined I_1 range. Each of these nests consists of a sequence of inner I_2 loops, one for each segment defined in that particular I_1 range. The individual bound expressions are chosen so as to delineate the segment. For the I_1 range $[1, n/2)$, three segments must be delineated. Accordingly, the I_2 axis is further split into ranges $[1, I_1]$, $(I_1, 2I_1]$ and $(2I_1, n]$. Similarly, the I_2 ranges can be found to delineate remaining segments in other I_1 ranges.

We use *Loops 1, 2* and *3* to compare the effectiveness of *CDA-guard-rem* and *Homogeneous-partitioning*. In Figure 4.14 darker bars correspond to the application of *CDA-guard-rem* and lighter bars correspond to the application of *Homogeneous-partitioning*. Algorithm *CDA-guard-rem* removed a significant number of guards in *Loops 1* and *3*. Hence, applying *CDA-guard-rem* may suffice when the intersection has a large number of iterations relative to the total number of iterations. When this is not the case, as in *Loop 2* then *Homogeneous-partitioning* is capable of removing a large proportion of the overhead. However, the performance differences between *CDA-guard-rem* and *Homogeneous-partitioning* will be further reduced when we apply techniques discussed in the next subsection to optimize the evaluation of the guards.

4.2.3 Optimizing the Evaluation of Guards

The run-time overhead of evaluating guards that may remain after having applied the guard elimination techniques described earlier can be further reduced by optimizing the clauses in the guards:

- Some of the guards may be redundant, in which case they can be removed. Clause c_j is redundant in guard $c_1 \wedge \ldots c_j \ldots \wedge c_k$, when $c_1 \wedge \ldots \wedge \overline{c_j} \wedge \ldots \wedge c_k \wedge C$ is false, where C is the conjunction of clauses corresponding to the bounds of the enclosing iterators. For

instance, in Figure 4.8, some of the clauses in the guards of the code generated for the range $k \leq I_1 \leq n$ and $0 \leq I_2 \leq k-1$ are redundant. Statement S_1 of the loop body has the guard $(0 \leq I_1) \wedge (I_1 \leq n) \wedge (0 \leq I_2) \wedge (I_2 \leq n)$. We can verify that the clauses $(0 \leq I_1)$ and $(I_1 \leq n)$ are redundant, since the enclosing I_1 iterator has a range of $k \leq I_1 \leq n$. Similarly, $(0 \leq I_2)$ and $(I_2 \leq n)$ are redundant. Hence, we can conclude that statement S_1 (and similarly S_2) does not need guards in the subnest considered.

- The guards for some statements may be inconsistent among themselves or with the bounds of the enclosing iterators, which implies that these statements are never executed. The overhead of guard computation for such statements can be eliminated by removing the statements altogether. When none of the guards in a subnest are consistent among themselves or with the bounds of the enclosing iterators, then the subnest only iterates through empty iterations, so the entire subnest can be removed. For instance, in the subnest of Figure 4.8 with $k \leq I_1 \leq n$ and $0 \leq I_2 \leq k-1$ statement S_2 is never executed. This is because the clause $k \leq I_2$ in the guard for statement S_2 is inconsistent with the bounds of the enclosing I_2 iterator.

- Guards can be further optimized if the clauses in the guards are all in canonical form, where the conditions on I_r only involve expressions in the enclosing iterators I_1 to I_{r-1}. This allows the conditions on iterator I_r to be evaluated and stored as boolean values before the beginning of the I_r iterations. The statement guards can then be reduced to a conjunction of previously computed boolean values. In general, it is beneficial to move the evaluation of the guards, to the outermost loop level possible.

4.3 Optimization of Space Overhead for Temporaries

The temporary variables introduced during Computation Decomposition may increase the number of references to memory and may add to space requirements and the cache footprint. A number of optimizations can reduce some of these overheads.

- Temporaries needed in a loop may be replaced by dead variables, which are not used in the later flow of the program.

- While decomposing a statement, it is possible to eliminate the need for temporary variables altogether by using the lhs array elements to store the intermediate results. Such a replacement

is legal if the dependence relations remain legal. Even though a Computation Decomposition does not modify dependences, eliminating the temporary variable this way can modify dependences. For example, it is legal to replace $t(i,j)$ by $a(i,j)$ in the following decomposition.

$$a(i,j) = a(i,j) + a(i-1,j) + a(i,j-1) \quad \Rightarrow \quad \begin{array}{l} \overset{a(i,j)}{\cancel{t(i,j)}} = a(i,j) + a(i-1,j) \\ a(i,j) = \underset{a(i,j)}{\cancel{t(i,j)}} + a(i,j-1) \end{array}$$

However, such a replacement would be illegal in the following decomposition, because $a(i,j)$ would be modified before it is used in the second statement so the temporary variable needs to be retained.

$$a(i,j) = a(i,j) + a(i-1,j) + a(i,j-1) \quad \Rightarrow \quad \begin{array}{l} t(i,j) = a(i-1,j) + a(i,j-1) \\ a(i,j) = t(i,j) + a(i,j) \end{array}$$

Hence, storage requirements can be reduced in this way for only some decompositions. Moreover, it must be noted that the dependences introduced by replacing the temporary variable can constrain later opportunities for Computation Alignment. It is therefore better to replace the references to the temporary by references to the lhs after the CDA transformation.

- Temporary variables that were been introduced can be reused in subsequent loops. This is possible since the temporaries are intended to store only the results inside a loop, and these results are not needed outside the loop.

- Temporary arrays are typically initially chosen to have the same dimension and size as the iteration space, since the subexpressions that generate values for the temporaries potentially have a new value in each iteration. The dimension and size of the temporary arrays can be reduced following the CDA transformation. It is only necessary to have as many storage locations as there are iterations between when the temporary is defined and when it is used. For simple offset alignments, the size of the temporary arrays can be just a fraction of the size of the iteration space. For example, consider the decomposition of a statement S in a two dimensional loop into statements S_1 and S_2. The results of S_1 are stored in a temporary array t. When statement S_1 is aligned to statement S_2 along the outer loop level by an offset c, then t need only be of size $c \times n$, assuming n iterations in the inner loop.

CHAPTER 5

Application of CDA to Reduce Number of Cache Conflicts

Arms on armour clashing bray'd
Horrible discord, and the madding wheels
Of brazen chariots rag'd: dire was the noise
Of conflict.
— *John Milton, Paradise Lost*

5.1 Reducing the Number of Cache Conflict Misses

In this chapter, we show the application of CDA to reduce the number of conflict misses in the cache. Cache conflict misses occur when different memory references map to the same location in the cache. Conflict misses can drastically reduce the cache hit ratio and thus significantly increase execution times. Therefore, reducing the number of cache conflict misses for improved cache utilization is an important optimization objective.

Traditional techniques used to reduce the number of cache conflicts modify the layout of arrays in memory. *Array padding* is one such technique, which modifies the array layout by manipulating the array sizes [5, 6]. *Cache partitioning* is a more recent technique, which modifies the array layout by introducing a suitable number of dummy locations between the arrays [39]. Bacon et al. describe an algorithm to derive a suitable array padding to remove cache conflicts in innermost loop iterations [5]. The cache partitioning algorithm, on the other hand, considers cache conflicts arising in all levels of loop nesting so as to remove cache conflicts between reusable portions of arrays. However, modifications to the array layout are global changes. The objective of this chapter is to show how CDA can be used as an alternative technique when global changes to the array layout are undesirable.

In order to determine the location of array elements in the memory and cache, we need to be able to represent the number of elements in an array between the first element and a chosen element of the array. For this purpose, we use an integer vector, $\vec{V} = (v_1, \ldots, v_m)^T$, called the *mapping*

vector such that v_i is the size of the m-dimensional array along array dimension $i+1$, and v_m is 1. Then, the array element accessed using reference matrix r in iteration $\vec{I}$ is $r\vec{I}\cdot\vec{V}$ array elements away from the first element of the array. As an instance, the mapping vector for a two-dimensional $n \times n$ array is $V = (n, 1)^T$ and reference $A(i, j)$ in iteration $(10, 5)$ accesses an element which is $10n + 5$ elements away from element $A(1, 1)$.

A conflict in a direct-mapped cache is depicted in Figure 5.1(a). An array access in iteration $\vec{I}$ with reference matrix R_1 maps to memory location $M_1 = C_1 + R_1\vec{I}\cdot\vec{V}$, where C_1 is a constant and $\vec{V}$ is the mapping vector. A second array access in iteration $\vec{I}$ with reference matrix R_2 maps to memory location $M_2 = C_2 + R_2\vec{I}\cdot\vec{V}$.[1] A cache conflict occurs for these two accesses if the cache geometry is such that both M_1 and M_2 map to the same cache line.

The cache conflict shown can be eliminated by suitably modifying the number of elements between M_1 and M_2, so that the accessed data elements map to different cache lines (Figure 5.1(b)). Modifying V changes the array sizes, and is referred to as intra-array padding; this is achieved by changing the declaration of the arrays to become larger along one or more dimensions. Modifying the C_i's changes the placement of the arrays in memory and is referred to as inter-array padding; this is achieved by inserting dummy variables between the array declarations.

The main idea behind using CDA to reduce the number of cache conflicts is to spread the conflicting accesses of an iteration into different iterations. While modification to array layout moves conflicting array accesses apart in *space*, CDA moves conflicting array accesses apart in *time*. In the example we are considering, the time (in number of iterations) between the access to M_1 and access to M_2 can be changed by aligning the statement containing R_2 relative to the statement containing R_1. In other words, the statements containing R_1 and R_2 can be aligned so that R_1 and the aligned R_2, R_2', do not access M_1 and M_2 in the same iteration; in iteration $\vec{I}$, R_1 would continue to access M_1, whereas R_2' would access a new location M_2'. In Figure 5.1(c), R_2 is changed to R_2' so that the new memory location $M_2' = C_2 + R_2'\vec{I}\cdot\vec{V}$ and M_1 map to different cache lines.

The following sections show how a CDA transformation can be efficiently derived that reduces the number of cache conflicts. It uses an algorithm that is similar to the one used to find appropriate array paddings [5]. In the next two sections, we describe a representation of cache conflicts and an algorithm that uses this representation to derive a CDA transformation. In the last section of this chapter, we compare the application of CDA and the application of array padding to reduce the

[1] Without loss of generalization we can assume here that arrays have the same size.

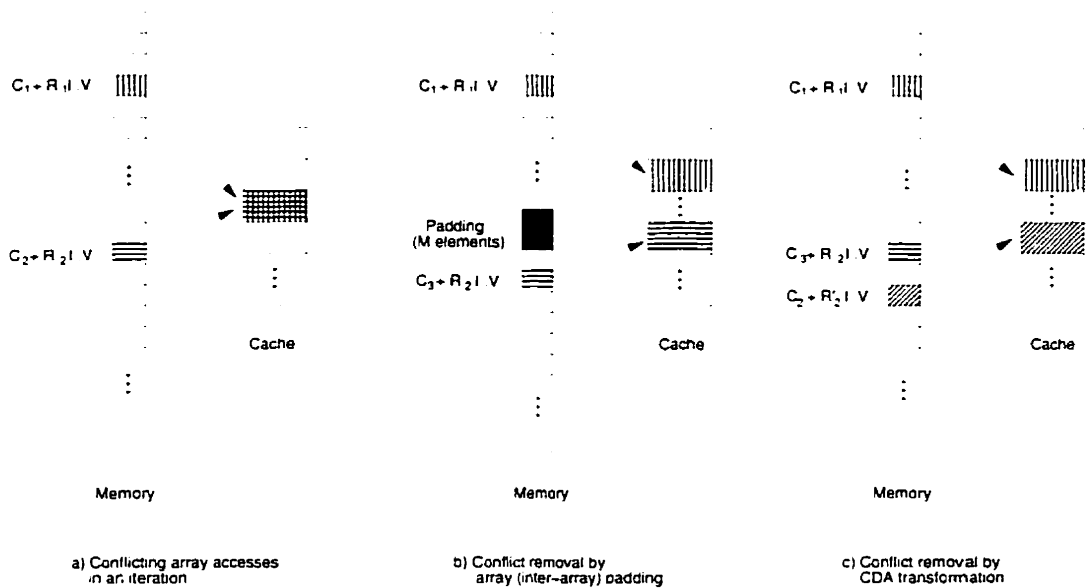


Figure 5.1: Reducing cache conflicts with modification to array layout and CDA.

number of cache conflicts. In Chapter 8, we provide experimental results comparing the effectiveness of these two techniques on example nested loops.

5.2 Representation of Cache Conflicts

The detection and representation of cache conflicts is, in general, complex. We represent the cache conflicts in a loop with a *conflict graph* $G = (V, E)$, where V is the set of array references in the loop and $(u, v) \in E$ when accesses to u and v result in a cache conflict for a given (i.e., first) iteration and for a given cache geometry. Although the conflict graph is a simple representation, it suffices to characterize the cache behavior with respect to cache conflicts. This representation is useful based on two main observations which padding algorithms have also exploited. First, cache conflicts are most expensive when they occur due to accesses in the same iteration; these should therefore be the main target of elimination. Second, iterations tend to be uniform in that they all have similar behavior with respect to conflict misses. Therefore, it suffices to represent the cache conflicts of just one iteration, such as the first. The uniformity ensures that the transformations based on analyzing these conflicts will typically be effective across all iterations. Figure 5.2 shows the conflict graph for a loop body containing the following statement:

$$S : A(i,j) = B(i,j) + B(i-1,j) + A(i-1,j) + A(i+1,j)$$

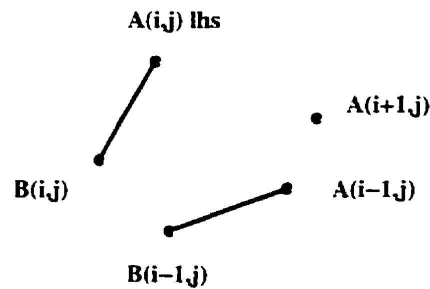


Figure 5.2: Conflict graph for statement S.

(assuming the data and cache sizes are such that the shown conflicts occur).

For this discussion, we assume that the cache is direct-mapped. This reduces the complexity of the decomposition problem and also makes the solution resilient to the order of references, patterns of variable reuse and dependence constraints on alignments. The derived CDA is still effective for other cache geometries, since accesses that do not usually conflict in a direct-mapped cache also do not conflict in caches of higher associativity. Moreover, the algorithm can be expanded to account for other cache geometries in a straightforward manner by modifying the steps to derive computation decompositions.

A CDA transformation of a loop modifies its conflict graph. Computation Decomposition can be viewed as a partitioning of the conflict graph, and Computation Alignment adds or removes edges in the graph as the references in partitions are aligned. The goal is to derive a CDA which minimizes the number of edges in the conflict graph.

5.3 Derivation of a Suitable CDA Transformation

A CDA transformation to reduce cache conflicts may be derived in a number of ways. *Bottom-up* and *top-down* represent two extreme approaches. In a bottom-up approach, statements might be decomposed down to the granularity of individual references before they are aligned. While this approach maximizes the search space for alignment, it also leads to an extreme number of alignment possibilities with correspondingly high search complexity; the transformed loop body invariably has more statements than necessary. In a top-down approach, the decomposition would be determined by a heuristic before applying Computation Alignment, but because a heuristic is being used, this approach may not expose all optimization opportunities.

In this section, we consider a "hybrid" approach: a heuristic is used to determine an initial decomposition as in the top-down approach, but statements may be further decomposed while

searching for alignments in order to increase the size of the search space when necessary. In this approach, the overhead of decomposing additional statements is incurred only when it is necessary to explore additional opportunities to reduce the number of conflicts.[2]

5.3.1 Initial Computation Decomposition

The number of ways a statement can be decomposed is exponential in the number of array references. Algorithm $A1$ of Figure 5.3 heuristically derives an initial decomposition of a statement, using a greedy approach. It partitions the conflict graph into *independent sets*, where a set contains references that only conflict with references in other sets. The number of independent sets is kept as small as possible so that the complexity of the subsequent alignment and the number of temporary variables needed is reduced. The algorithm does not attempt to partition the conflict graph into the minimum number of independent sets, because the problem of finding the *achromatic number* or the minimum number of independent sets of a graph is NP-complete [20].

Each iteration j of Step 2 extracts a *maximal* independent set V_j from the remaining vertices of the conflict graph. Step 3 chooses the first vertex of this set as the one with the smallest degree (i.e., having the fewest conflicts) from the remaining vertices of the conflict graph. Step 4 constructs the subset, U, of all vertices which do not have a common edge with any vertex in V_j. The members of U are all candidate vertices that may be added to V_j. The vertex in U with the smallest degree is chosen and added to V_j; at the same time it is removed from the conflict graph. When U is empty, then there are no more vertices to be added to V_j. Vertices with smallest degree are chosen from U (and added to V_j) because this increases the chances that more vertices can be added to V_j in subsequent steps, since there are a larger number of vertices left outside V_j which do not have an edge with the vertex with smallest degree.

Algorithm $A1$ partitions the conflict graph of Figure 5.2 as shown in Figure 5.4. The decomposed loop body corresponding to this partitioning is:

$$
\begin{aligned}
S_1 &: \ t(i,j) = B(i,j) + B(i-1,j) \\
S_2 &: \ A(i,j) = t(i,j) + A(i-1,j) + A(i+1,j)
\end{aligned}
$$

The references in each of the statements are conflict free, and hence the conflicting references of the original statement were distributed into the two statements.

[2] We continue to assume that the array references are affine functions of the iterators. Thus, the subscript functions may be coupled, although they tend to be simple when cache conflicts occur uniformly in all iterations. The subscripts in some array dimensions may be independent of the iterators. The array and loop dimensions may be different.

Algorithm : $A1$
input: Statement S
output: Initial Computation Decomposition
begin
1. $G \leftarrow (V, E)$, such that
 $V \equiv$ references in S, and
 $(u, v) \in E$ iff u and v conflict in cache
 $j = 0$
2. *loop*
 $j \leftarrow j + 1$
3. $V_j \leftarrow \{v\}$ where $v \in V$ has smallest degree
4. *repeat*
 $U \leftarrow \{w \mid \forall u \in V_j, (u, w) \notin E\}$
 $r \leftarrow$ reference with smallest degree in U
 $V_j \leftarrow V_j \cup \{r\}$
 $V \leftarrow V - \{r\}$
 until $U = \phi$
 end loop
end

Figure 5.3: Algorithm $A1$ derive initial Computation Decomposition.

Note that the conflict graph does not have vertices corresponding to temporary array references. This is because temporary variables are either re-indexed or chosen to have suitable size so that they do not introduce new set of conflicts. Thus, Computation Decomposition itself does not introduce additional constraints on Computation Alignment.

5.3.2 Deriving the Computation Alignment

After the initial decomposition, we search for the Computation Alignment that minimizes the number of conflicts. We consider only integer offset vectors for alignment. This has two advantages:

i) The search for offset vectors is significantly simpler than a search for *all* alignments. Considering *all* possible alignments would be inefficient, since potentially all $(n + 1)$-dimensional non-singular integer matrices would have to be examined for each of the statements in the loop body. In contrast, the search for offset vectors involves the search for only n integer values for the elements of the vector.

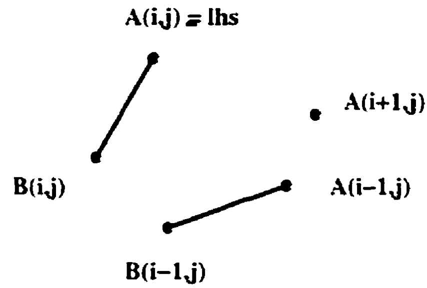


Figure 5.4: Initial decompositions of conflict graph for statement S.

Alignment by offset vectors can be as effective as general non-singular integer matrices in reducing conflicts. This is because the objective of a transformation is to modify the distance (in time) between conflicting accesses. It is not necessary that this distance be a general linear function of iterators.

ii) Alignment by integer offset vectors produces more efficient transformed code than alignment by general non-singular integer matrices: the loop bound expressions and the array references tend to be simpler, and guard elimination techniques can be more effective with the majority of the iterations executing all statements.

In order to show how integer offset alignments can be found, consider again the initial decomposition of statement S:

$$\begin{aligned} S_1 &: \; t(i,j) = B(i,j) + B(i-1,j) \\ S_2 &: \; A(i,j) = t(i,j) + A(i-1,j) + A(i+1,j) \end{aligned}$$

To reduce the number of cache conflicts, the array accesses in S_1 can be moved in time along the i iterator relative to array accesses in S_2. Assuming a row-major storage order, our first attempt should be to align computations $(i+1,j;S_1)$ to $(i,j;S_2)$, which changes reference $B(i,j)$ in S_1 to $B(i+1,j)$ and reference $B(i-1,j)$ to $B(i,j)$. With this alignment, the statements continue to have conflicts, namely between $B(i+1,j)$ and $A(i+1,j)$ and between $B(i,j)$ and $A(i,j)$, so further alignment is necessary. Continuing in this fashion, all conflicts can be eliminated in this case by shifting the S_1 computations further, aligning $(i+3,j;S_1)$ to $(i,j;S_2)$. This transformation produces the new loop body:

$$S_1 : t(i+3,j) = B(i+3,j) + B(i+2,j)$$
$$S_2 : A(i,j) = t(i,j) + A(i-1,j) + A(i+1,j)$$

Note that the subscript functions in the references to the temporary t may have to be changed suitably so as to not introduce any conflicts. An alternative is to choose a non-conflicting size for t.

Statements S_1 and S_2 can also be aligned along the the j iterator to eliminate the conflicts. In this case, however, it turns out that the value of the offset has to be the number of array elements in a cache line.

This technique to find a suitable Computation Alignment for reducing the number of cache conflicts is similar to the technique used by padding algorithms. In order to eliminate the conflict in statement S, padding algorithm would typically add a dummy row between A and B, but this will just cause conflicts between $A(i+1,j)$ and $B(i,j)$ and between $A(i,j)$ and $B(i-1,j)$, since $B(i,j)$ and $B(i-1,j)$ are now mapped to locations previously occupied by $B(i+1,j)$ and $B(i,j)$, respectively. Adding two more dummy rows between A and B removes all conflicts.

Algorithm $A2$ of Figure 5.5 uses this technique to search for suitable offset alignments for each statement so as to minimize the number of cache conflicts. An exhaustive search for alignments is not practical for larger loop dimensions and loop bodies, since there are exponentially many possible offset alignments. For example, with K statements in the decomposed loop body and C integer offsets for each of the n dimensions of legal offset vectors, we would need to examine C^{nK} alignments. Moreover, C can be very large in practice. Hence, algorithm $A2$ uses techniques similar to those of array padding algorithms to heuristically align statements in a more efficient way. The statements are aligned in some sequence, and once a statement has been aligned, it is no longer changed. This is similar to how padding algorithms work: the padding between two arrays is not changed once set, and especially not due to paddings between other arrays that are set later. This is a greedy approach which is polynomial, but it may not produce an optimal solution in that it may not be able to eliminate all conflicts. Our algorithm refines decompositions at the point where it is determined that it is not possible to eliminate all conflicts with this heuristic and the original decomposition.

In algorithm $A2$, each iteration of the outer while-loop of Step 1 derives an integer offset vector a for a statement S_i of the loop body. The order in which statements are considered is not important, since the transformations are relative and decompositions are iteratively refined. Initially, all elements of vector a are zero, implying no alignment. Step 2 creates a *list* of candidate

```
Algorithm: A2
input: Decomposed loop body with statements S_1 ... S_K
output: Integer offset alignment for each statement
begin
1.  while there is a statement S_i to align
        a^T <- [a_1, ..., a_n, 1], with a_j = 0, ∀j
        T_i <- [I | a]        // T_i is an alignment transformation for S_i
        S_aux <- ∅
2.      Construct a list of candidate iterators that appear in the array references of S_i.
3.      Order the iterators in the list.
4.      for each i_j in the list
5.          while conflicts exist due to S_i references using i_j
6.              while T_i is legal and conflicts exist due to S_i references using i_j
                    a_j <- a_j - 1
                end while
                T_i <- [I | a]
7.              if T_i is illegal then
                // attempt offsets in opposite direction
                    a_j <- 0
                    while T_i is legal and conflicts exist due to S_i references using i_j
                        a_j <- a_j + 1
                    end while
                end if
                T_i <- [I | a]
8.              if T_i is illegal then
                    a_j <- 0
                    T_i <- [I | a]
                    R <- set of references in S_i whose dependences are violated
                    if R contains all references in S_i then
                        break out of innermost while-loop
                    else
                    decompose S_i so that S_aux contains all references in R
                    end if
                end if
            end while
9.          if all conflicts w.r.t. S_i references are eliminated then
                break out of for-loop
            end if
        end for
    end while
end
```

Figure 5.5: Algorithm $A2$ to derive alignments that reduce cache conflicts.

iterators for alignment. The *list* contains only those iterators used in the array references of the current statement S_i. We need to consider only these iterators, since the data elements accessed in a loop are invariant across iterators absent in the array subscript functions.[3] From this *list*, we also exclude those iterators that are useful in eliminating cache conflicts.[4] An iterator i is not useful in eliminating conflicts if for every reference r in statement S_i, array elements $r\vec{I}$ and $r'\vec{I}$ map to the same cache line, when r' is obtained by replacing every occurrence of i in r by $i+1$ and $\vec{I}$ is an iteration (such as the first).[5]

Step 3 *orders* the iterators in the *list* heuristically, taking into account such factors as parallelism, variable reuse and the sizes of the temporary arrays. For example, ordering the list from innermost iterator to outermost iterator tends to preserve outer loop parallelism, since the dependences that alignment may introduce will then tend to be at the inner loop levels. This ordering also reduces the sizes of the temporary arrays. When the target architecture is a uniprocessor, ordering from outermost iterator to innermost tends to improve data reuse. This is because alignment along the outer iterators moves the conflicting accesses farther apart in time than if the alignment were along the inner iterators. Such an ordering increases the probability that an array reference involved in the conflict is not moved to an adjacent iteration where it conflicts with previously accessed data elements that may be reused. When both data reuse and parallelism are important, as on shared memory systems, then a combination of the above two orders should be applied. Aligning the non-parallel iterators preserves parallelism; and among the non-parallel iterators, aligning the outermost iterator first improves data reuse. It should be noted, however, that the order of iterators in the *list* does not affect the ability to remove cache conflicts.

For each iterator i_j in the *list*, the loop of Step 5 considers all legal offsets to determine whether they remove conflicts. The search stops when there are no conflicts due to S_i references or when dependences prevent further alignment. Both negative and positive offsets are considered. The positive offsets move the computations of S_i earlier in time with respect to dimension i_j, while the negative offsets move the computations of S_i later in time with respect to dimension i_j. Step 6 iteratively tries larger and larger negative offsets along iterator i_j, until there are either no more conflicts due to the S_i references using i_j or the alignment becomes illegal. The offset is

[3]For instance, in a loop with i, j and k iterators and a reference $A(i,k)$, only alignments along i and k have an effect. Clearly, cache conflict between array references that use only constants, such as $A(1,1)$ and $B(1,1)$ cannot be eliminated as the same elements are accessed in all iterations, and array padding would have to be used in this case.

[4]We need to eliminate these, to prevent infinite iterations in the algorithm.

[5]When iterator i indexes into the last dimension of the array referenced by r, then r' is obtained by replacing every occurrence of i in r by $i+l$, where l is the number of array elements in a cache line.

generally decremented by 1 at a time, but as an optimization, the offset could be decremented by the size of the cache line when the iterator indexes into the last dimension of an array. When the transformation T_i being considered is illegal, then larger and larger positive offsets are tried in Step 7. Generally, the offsets required in Steps 6 and 7 are typically small, just as padding typically requires only a small constant number of rows/elements, so that only few iterations of these steps should be necessary.

There are two cases to consider in Step 8 when Steps 6 and 7 cannot find a legal alignment T_i. When dependences are violated for *all* references in S_i, then we revert back to a zero offset by breaking out of the while loop (to try offsets along other iterators in the *list*). The second case to consider is when T_i violates dependences for only *some* references in S_i, but is legal for the remaining references. In this case, we refine the decomposition to isolate those references that cause T_i to be illegal into a newly created statement S_{aux}. Statement S_{aux} is aligned separately in a later iteration, and the search for an appropriate alignment for the now smaller statement S_i continues. (It may be necessary to continue, because the illegal T_i may have stopped iterations of Steps 6 or 7 before all the conflicts were eliminated).

As an example of refining a decomposition, consider the alignment of statement S which was decomposed earlier into the following statements S_1 and S_2:

$$\begin{array}{ll} S_1: & t(i,j) = B(i,j) + B(i-1,j) \\ S_2: & A(i,j) = t(i,j) + A(i-1,j) + A(i+1,j) \end{array}$$

Aligning computations $(i+1, j{:}S_1)$ to $(i, j{:}S_2)$ changes reference $B(i,j)$ to $B(i+1,j)$ and reference $B(i-1,j)$ to $B(i,j)$. Suppose this violates dependences between S_1 and another statement in the loop body, say due to reference $B(i-1,j)$. Step 8 then decomposes S_1, so that the new statement S_{aux} contains $B(i-1,j)$.

$$\begin{array}{ll} S_{aux}: & t2(i,j) = B(i-1,j) \\ S_1: & t(i,j) = B(i,j) \\ S_2: & A(i,j) = t(i,j) + t2(i,j) + A(i-1,j) + A(i+1,j) \end{array}$$

At this point, there are still conflicts between references $A(i,j)$ and $B(i,j)$ and references $A(i-1,j)$ and $B(i-1,j)$. Therefore, the next iteration of Step 5 aligns S_1 so as to change reference $B(i,j)$ in statement S_1 to reference $B(i+2,j)$, effectively eliminating the conflict between $A(i,j)$ and $B(i,j)$. Statement S_{aux} is then later aligned in a subsequent iteration of Step 1, along the j iterator. One

such alignment along the j iterator will change reference $B(i-1, j)$ to reference $B(i-1, j+l)$, where l is the size of the cache line.[6]

The initial decomposition in Algorithm *A1* ensures that there are no conflicts between references within any single statement of the decomposed loop body. This allows each statement to be applied as large an offset alignment as required to remove the conflicts. On the other hand, if the original loop has dependences, then *A1* and *A2* may only reduce the number of conflicts, but not remove them all. The impact of dependences on the algorithms can be summarized as follows:

i) CDA transformations derived for loops with dependences will generally have more decompositions than the transformations derived for loops without dependences. This is because any decompositions will be further refined when a candidate alignment violates a dependence.

ii) Dependences with larger elements in the distance vectors allow larger offset alignments. This is because larger elements imply that the dependent iterations are farther apart in time, so that there are a larger number of iterations into which array accesses can be moved.

iii) While aligning decomposed statements, flow dependences are stricter constraints than anti-dependences. With an anti-dependence, the read access can be moved to any iteration before the iteration in which corresponding write access occurs, whereas with a flow dependence, the earliest a read access to an array element can be is in the iteration in which the element is written.[7]

iv) A flow dependence in the innermost dimension, such as between references $A(i, j)$ and $A(i, j-1)$ in a 2-dimensional loop with j as its inner iterator, is a very strict constraint. This is because the only alignment possible (other than the identity transformation) modifies $A(i, j-1)$ to $A(i, j)$. On the other hand, a dependence between $A(i, j)$ and $A(i-1, j)$ has several legal alignments along the j iterator (and only one along the i iterator).

5.4 Comparison of CDA with Padding

Modification of the array layout using padding is relatively easy to implement and can be very effective. However, it does have a number of drawbacks, and CDA can be used as an alternative

[6] Note that if dependences due to all references in S_1 were violated, then Step 4 would align S_1 along the j dimension.

[7] With a flow dependence, the read access in a decomposed statement cannot be moved later in time, since doing so would violate the flow dependence on the temporary variables.

technique when these drawbacks prevail. Some of the drawbacks of modifying the array layout are:

- It changes the data declarations, so it has a global effect: an effective padding for one loop can introduce interferences in another loop.
- There are situations where modifying the array layout is illegal such as when the program accesses an array in a shape different than the one declared. For instance, n^2 elements may be accessed in the same program both as an $n \times n$ 2-dimensional array and as a n^2 element linear array with another name. These situations are difficult to identify without integrated support for inter-procedural analysis.
- Array padding cannot be adapted to suit a range of data sizes, since it shifts the data size for which interferences occur by just manipulating array sizes. Therefore, it is not possible to ensure conflict-free cache behavior for a range of data sizes.

CDA can be employed in place of and in conjunction with the previously existing techniques. In this regard, CDA has some advantages:

- It makes only local changes, so unlike padding, it does not introduce additional global constraints.
- Since it does not change the data declarations, programs relying on the original layout of arrays will work correctly.
- To support data sizes that are unknown at compile time, it is possible to generate multiple CDA-transformed loops and then dynamically choose an appropriate one at run-time when the data sizes are known. Such an option is not possible with padding.

The application of CDA to reduce the number of cache conflicts also has some disadvantages:

- CDA transformations may not be able to eliminate all the cache conflicts in the loop in the presence of dependences.
- CDA transformation may not eliminate cache conflicts that occur in the outer iterations of the loop.
- CDA transformed loops require additional storage space for the temporaries and in fact increase cache footprint. The transformed loop also incurs overhead due to references to

temporary array elements (which are designed to be cache hits). However, these overheads can be optimized as described in Section 4.3.

- CDA transformed loops can be much larger than the original loop and hence they can take longer to compile.

Overall, our experiments of Chapter 8 show that loops with cache conflicts removed through array padding tend to run somewhat faster than code with cache conflicts removed through CDA transformations because of the overheads CDA introduces. Nonetheless, CDA is effective in significantly improving the run-time of loops with cache conflicts, as it is useful for those cases where array padding cannot be usefully employed.

CHAPTER 6

Application of CDA to Remove Ownership Tests from SPMD Codes

> I pass by that way in the gloaming with Mary;
> 'I wonder,' I say, 'who the owner of those is'.
> 'Oh, no one you know,' she answers me airy ...
> — *Robert Frost, Asking for Roses*

In a parallel programming environment, such as for HPF [18], data and computations are mapped onto the processors in two steps. First, either the user or an automatic tool aligns and maps the arrays onto the processors. Then, the computations are mapped onto the processors depending on how the data was mapped. The compiler generates a *single* SPMD (Single Program Multiple Data) program that is to be executed by *all* processors. This program uses a *computation rule* to dynamically determine at run-time which computations to execute on which processors, given the data they own. One computation rule, which is used almost exclusively, is the *owner-computes* rule. With this rule, a statement is executed on the processor that owns the lhs data element of the statement. For this purpose, an intrinsic function is needed for each processor to execute the tests for ownership in order to decide whether a statement in an iteration should be executed or not [18, 21]. Thus, the efficiency of SPMD code depends in part on:

(*i*) the number of non-local accesses each processor must perform, and

(*ii*) the overhead of the chosen computation rule; in particular, how often ownership tests must be executed.

By choosing a computation rule other than owner-computes it may be possible to improve the efficiency of SPMD code significantly by reducing the number of non-local accesses. For this purpose, we first define *flexible* computation rules in this chapter, called *P-computes rules*. P-computes rules are more general than the owner-computes rule in that they consider the location of all the data needed for a computation instead of just the lhs. For simplicity, we also refer to the intrinsic functions used in implementing flexible computation rule as *ownership tests*. We then show

how the efficiency of SPMD code can be further improved by reducing the number of ownership tests using CDA transformations.

It is possible to improve the efficiency of SPMD code in a separate and later phase by additional optimizations such as inserting collective communications. The efficiency of SPMD code ¿ can also be improved by applying *subspace* analysis developed recently by Knobe [25]. In this context, CDA can be viewed as a transformation capable of reducing the *natural subspace* of the loop body.

In the first section, we introduce P-computes rules, which can be used to map computations onto processors so as to minimize communication. In Section 6.2, we describe how ownership tests can be removed by using data alignment and CDA transformations. In Sections 6.3 and 6.4 we describe an algorithm to derive CDA transformations that remove ownership tests in SPMD programs which use P-computes rules. In Chapter 8, we provide experimental results to show the effectiveness of applying CDA to remove ownership tests.

6.1 *P-computes*: Flexible Computation Rules

Computation rules, such as the owner-computes rule, can be called *fixed computation rules* in that:

(*i*) they do *not* consider the locations of *all* the data elements accessed in the computations of a statement,

(*ii*) the rules are applied at the granularity of entire statements, and

(*iii*) all the statements in a program are (typically) mapped using the same rule.

Fixed computation rules do not result in optimal mapping of computations because the fixed computation rules do not consider the locations of *all* the data elements accessed in the computation. For example, with the owner-computes rule, the computations of a statement are executed by the owner of the array element on the lhs of the statement, even though fewer non-local accesses may be required if executed by the owner of an array element being accessed on the rhs of the statement. The appeal of fixed computation rules is that they are simple, and not that they are communication optimal.

Flexible computation rules, on the other hand, are more general than fixed computation rules since:

(*i*) they can consider the locations of *all* the data elements accessed in the computations in question,

(ii) they can map individual sub-expressions onto processors and not just entire statements.

Flexible computation rules provide more opportunities for optimizing communication, because a computation can be mapped onto the processor which owns most of the data elements needed to execute the computation. Therefore, flexible computation rules are significantly more powerful than fixed computation rules.

In this section, we introduce flexible computation rules called P-computes rules.[1] These rules are specified by $\odot$ operators, which are inserted into the code either by the programmer or by a restructuring compiler. An expression with the $\odot$ operator, $\odot_p\ (expr)$, specifies that the expression $expr$ is to be executed on processor p. In general, p and $expr$ are functions of the enclosing loop iterators, and $expr$ may also contain other $\odot$ operators.

In $\odot_p\ (expr)$, function p is typically chosen so as to minimize communication when executing $expr$ in the current iteration. If $expr$ is an entire statement, then the result of rhs is to be sent from the processor it is executed on, namely p, to the owner of the lhs who performs the and assignment. If $expr$ is a sub-expression, then the result of executing $\odot_p(expr)$ is sent from the processor designated by p to the processor designated by the enclosing $\odot$ operator.

The specification of p in $\odot_p(expr)$ can be either *direct*, where the identity of a processor is directly specified as a function of the iterators, or *indirect*, where the processor is indirectly specified in terms of the location of array elements. Direct specification is powerful, but non-intuitive. Indirect specification makes the flexible rules a natural extension to the owner-computes rule. It also enables reasoning about the mapping of computations relative to the array elements used to specify mapping. For instance, if the array elements used to specify the rule are co-located, then we can reason that the computations will be executed on the same processor as well.

Indirect specification is achieved through an intrinsic, $owner(e)$, which returns the processor that owns data element e. This intrinsic is similar to $iown(e)$, which is used by the owner-computes rule and evaluates to *true* on the processor that owns data element e and false on all others. In fact, $iown(e)$ is equivalent to the conditional $(myid = owner(e))$.

The $\odot$ operator is very general and can be used to express a variety of computation rules. Consider the following examples:

$$\odot_{owner(A(i))}(A(i) = A(i) + B(i))$$
$$\odot_{(i\ div\ b1)}(A(i) = \odot_{(i\ div\ b2)}(B(i) + \odot_{(i\ mod\ P)}(C(i) + D(i))))$$
$$\odot_{owner(A(i))}(A(i) = \odot_{owner(B(i))}(B(i) + C(i)) + \odot_{owner(C(i))}(C(i) + D(i)))$$

[1] P is for Processor.

The first example implements a rule equivalent to the owner-computes rule. The second example specifies the processors directly as a function of the loop iterator, whereas the mapping is indirect in the last example.

The techniques presented in this chapter assume that the owners of computations are specified indirectly. For brevity, we sometimes use $\Im_{A(I)}$ to mean $\Im_{owner(A(I))}$.

The derivation of the optimal computation rule is computationally hard (although it is possible to derive optimal computation rules automatically for restricted classes of expressions and machine topologies [13]). A programmer usually has much better insight into the application, so she is most suitable for specifying the rules. It is possible to derive optimal computation rules automatically for restricted classes of expressions and machine topologies [13].

The choice of which flexible computation rule to use is often a trade-off between the opportunities for communication optimization and the simplicity with which SPMD code can be implemented. For example, generating efficient SPMD code with flexible computation rules is more complex than with fixed computation rules: the removal of ownership tests becomes more challenging, since each sub-expression may be mapped differently. In the section that follows we show how CDA can be used to reduce the intrinsics associated with P-computes rules. The techniques are clearly also applicable in the context of a fixed computation rule such as owner-computes.

6.2 Removing Ownership Tests in P-computes rules

In this section, we show how ownership tests in P-computes rules can be removed, first by using data alignment and then by CDA transformations. To show how the overhead associated with ownership tests can affect the efficiency of SPMD code, consider the loop on the left hand side of Figure 6.1. The computations are to be mapped onto P processors numbered from 0 to $P-1$. Assume that the arrays are aligned so that $A(i,j)$ and $B(i,j)$ are co-located, and that the arrays are distributed so that rows i of A and B are mapped onto processor $i\ div\ b$, and $b = \lceil \frac{n}{P} \rceil$. Each of the P processors contain b consecutive rows of A and B.[2] The indirect P-computes rule in this case happens to be equivalent to the owner-computes rule. Each processor executes every iteration, and a processor executes S_1 or S_2 only when it owns the lhs array elements.

This SPMD code is inefficient (even if it minimize communication), since a majority of the

[2]The boundary processors 0 and $P-1$ may contain fewer rows. For simplicity, we will assume here that P divides n evenly.

```
for i = 1,n
  for j = 1,n
    S1 : ∃owner(A(i,j)) ( A(i,j) = ... )
    S2 : ∃owner(B(j,i)) ( B(j,i) = ... )
  end for
end for
```

```
align B(j,i) to A(i,j)
...
for i = p*b,(p+1)*b-1
  for j = 1,n
    S1 : A(i,j) = ...
    S2 : B(j,i) = ...
  end for
end for
```

```
for i = p*b,(p+1)*b-1
  for j = 1,n
    S1 : A(i,j) = ...
    S2' : B(i,j) = ...
  end for
end for
```

Figure 6.1: SPMD codes for an example loop.

iterations executed by the processors do not need to execute either the S_1 or S_2 computations. In particular, processor p has computations to execute in only $2bn - b^2$ of the n^2 iterations:[3] in the other iterations it executes only ownership tests. Out of the $2bn - b^2$ iterations, only b^2 iterations have both S_1 and S_2 computations. Some $bn - b^2$ iterations have only S_1 computations and the remaining $bn - b^2$ iterations have only S_2 computations.

This SPMD code can be optimized at compile-time to minimize overhead at run-time. Our objective is to modify the code so that:

(i) all computations of an iteration are to be performed by the same processor, and

(ii) a processor examines only those iterations where it is guaranteed to find work.

In the ideal case, when these objectives have been achieved, then a processor will execute only those iterations that have been allotted to it, without requiring any ownership tests at all. These objectives can often be achieved through appropriate data transformations or code transformations.

Consider first removing ownership tests through data alignment. The arrays referenced in the lhs of the statements can be aligned (or re-aligned) so that the same processor owns all lhs elements of an iteration. The loop bounds can then be modified so that each processor scans only the subset of the iteration space that contains the lhs data elements local to the processor.

The SPMD code on the left hand side of Figure 6.1 can be optimized to the code in the middle of the figure using proper data alignments. In this case, arrays A and B were re-aligned so that $B(j,i)$ and $A(i,j)$ became co-located for every i and j in the array bounds. Using the function that maps the rows of the arrays onto processor, the lower and upper loop bounds are derived so as to scan only the iterations that access local elements of the lhs arrays. In this case, the lower and upper bounds for processor p are $p * b$ and $(p+1) * b - 1$, respectively.

[3] Processor p has computations to execute only in iterations $(i,*)$, where $i \ div \ b = p$, and in iterations $(j, *)$, where $j \ div \ b = p$.

It should be pointed out, however, that data alignment cannot remove ownership tests when multiple lhs references in an iteration are to the same array. Also, data alignment has the drawback that it is a global transformation: it can interfere with and constrain data alignment for optimizing communications. While it is possible to re-align data at run-time, it is expensive to do so, since it involves data movement.

CDA transformations can be used as an alternative to and in addition to data alignment, since there is a duality in aligning computation spaces and aligning arrays. Computation spaces can be aligned without changing existing data alignments, and so it is not necessary that all lhs data elements in the new iterations are owned by the same processor. The loop on the right hand side of Figure 6.1 is a result of a Computation Alignment. In this case, the computation space for S_2 was transposed with respect to the computation space for S_1 so that $A(i,j)$ and $B(i,j)$ are accessed in the same iteration. Moreover, the loop bounds were modified so that each processor scans only its local iteration space.

The main advantage of using a CDA transformation is that it is possible to align computation spaces of two statements, even when they reference the same array on the lhs, something that cannot be achieved through data alignment. However, it should be pointed out, that it is not always possible to find a legal CDA transformation capable of removing ownership tests. Hence, CDA must be viewed as a complementary transformation that can be used in conjunction with data alignment.

The general strategy to remove ownership tests is to modify the SPMD code so that, in the ideal case, each processor executes all computations in all of the iterations it executes. Towards this goal, we transform loops in two steps:

(i) *Packing iterations* — we use CDA transformations to minimize the number of processors that need to execute computations of an iteration. This allows us to coalesce the ownership tests of an iteration into fewer tests; they can be coalesced into one test in the ideal case when all computations of an iteration are to be executed by the same processor.

(ii) *Scanning local iteration space* — we use a technique similar to the guard elimination technique described in Section 4.2 that allows each processor to execute its local iteration space. In many cases, one can eliminate the need to execute ownership tests entirely by appropriately modifying the loop bounds as was done for the loop in Figure 6.1.

6.3 Derivation of CDA Transformation to Pack Iterations

In this section, we describe an algorithm capable of deriving a CDA transformation that packs iterations in such a way that the computations of an iteration belong to as few processors as dependences permit. We assume that the P-computes operators are specified in the target loop. The CDA transformation is derived in three stages:

(i) The target loop is decomposed such that each sub-expression enclosed by a P-computes operator becomes a separate new statement.

(ii) The statements of the decomposed loop body are computationally aligned so that the statements are mapped onto a single processor if possible; when dependences prevent such an alignment, then the statements are mapped onto as few processors as possible.

(iii) Finally, temporary arrays, if they exist, are data aligned to the other arrays on the lhs so that the computation rules for the statements with temporary lhs can be specified in terms of the temporary arrays.

The three stages are described in the following subsections. We continue to assume that the references are affine functions of the iteration vector $\vec{I}$ and are represented by reference matrices. In order to simplify our notations, the function f in an array reference $A(f(\vec{I}))$ is also used to denote the reference matrix. Data alignments also have a matrix representation: the data alignment transformation of an array, represented by non-singular integer matrix d_j, changes each reference matrix r of the array into reference matrix $d_j r$.

6.3.1 Derivation of Computation Decomposition

Algorithm *B1* in Figure 6.2 decomposes a statement S in the loop body such that after the decomposition each $\odot$ operator applies to a separate statement. Algorithm *B1* is applied to each statement of the loop body, with the statements being selected in an arbitrary order.

In algorithm *B1*, variable i is a counter for the number of statements generated as a result of decomposing S. In each iteration of Step 2, we choose an innermost $\odot$ operator-expression of S and make it a new statement with a new temporary array element as its lhs (Steps 3–5). In Step 6, the rhs of statement S is modified so that the sub-expression chosen in Step 3 is replaced by the corresponding reference to the temporary. In Step 7, statement S is replaced by statement S_i, where the rhs has been modified by the decompositions of Step 2.

Algorithm : *B1*
input: An arbitrary statement S : $\odot_{owner\ (A(f(\vec{I})))}$ $(lhs = rhs)$ of the loop body.
output: Decompose S at $\odot$ operators.
begin
1. $i \leftarrow 1$
2. *while* rhs of S has a P-computes operator
3. Choose an innermost $\odot_{owner\ (B(g(\vec{I})))}$ $(expr)$ in S
4. $t_i \Leftarrow$ new temporary array
5. generate statement S_i : $\odot_{owner\ (B(g(\vec{I})))}$ $(t_i(\vec{I}) = expr)$
6. replace $\odot_{owner\ (B(g(\vec{I})))}$ $(expr)$ in rhs by $t_i(\vec{I})$
   $i \leftarrow i + 1$
   end while
7. generate S_i : $\odot_{owner\ (A(f(\vec{I})))}$ $(lhs = rhs)$
end

Figure 6.2: Algorithm *B1* to decompose statements.

As an example, consider the $\odot$ operators in the loop on the top half of Figure 6.3. Algorithm *B1* decomposes the only statement of the loop into two statements so that each new statement has a single $\odot$ operator. A temporary array t is used to store the intermediate results. The decomposed loop is shown in the bottom half of the figure.

6.3.2 Derivation of Computation Alignment

Algorithm *B2* of Figure 6.4 searches for computation alignments for each of the statements generated by algorithm *B1*. We assume that the result of applying algorithm *B1* to the original loop is a new loop body with K statements. The objective is to pack iterations in such a way that the computations of an iteration belong to as few processors as dependences permit. The search space of all legal computation alignments is large, so it is not possible to exhaustively search this space in reasonable time. Unfortunately, however, it is also not sufficient to limit the search to offset alignment vectors in this case, as was done when reducing cache conflicts. This is because the statements may have reference matrices that would require non-singular transformations to transform the statements. In algorithm *B2*, we restrict the number of candidate alignments by focusing the search to alignments that make the array references in the $\odot$ operator-expressions similar.

Algorithm *B2* first attempts to map all statements onto a single processor (Steps 1–5), and

```
for i = 2, n
  for j = 2, n
    S1 : ⊙_A(i,j) [ A(i,j) = ⊙_A(i-1,j) [ A(i-1,j) + B(i-1,j) + C(i-1,j)+
           A(i-1,j-1) + B(i-1,j-1) + C(i-1,j-1) ]+
           A(i,j-1) + B(i,j-1) + C(i,j-1) ]
  end for
end for
```

```
for i = 2, n
  for j = 2, n
    S1.1 : ⊙_A(i-1,j) [ t(i,j) = A(i-1,j) + B(i-1,j) + C(i-1,j)+
                     A(i-1,j-1) + B(i-1,j-1) + C(i-1,j-1) ]
    S1.2 : ⊙_A(i,j) [ A(i,j) = t(i,j) + A(i,j-1) + B(i,j-1) + C(i,j-1) ]
  end for
end for
```

Figure 6.3: Computation Decomposition of a loop with $\odot$ operators.

then, if necessary, attempts to map the statements onto as few processors as dependences allow (Steps 6–12).

Step 2 constructs K sets of possible candidate alignments for the K statements in the loop body. The first set aligns each statement to S_1, the second set aligns each statement to S_2, and so on.[4] The set of transformation matrices that align each statement to statement S_i is denoted by α_i. The set of all α_i, $1 \leq i \leq K$, is denoted by $\mathcal{A}_{all}$. Each transformation matrix is derived similar to the way the transformation matrix is derived for element-wise data alignment. Consider the two statements:

$$S_i : \odot_{owner\ (\ A_i(f_i(\vec{I}))\ )}$$
$$S_j : \odot_{owner\ (\ A_j(f_j(\vec{I}))\ )} ,$$

where A_i and A_j are different arrays. The objective is to modify the statements so that the intrinsic *owner* function in both statements evaluates to the same processor. An element-wise data alignment achieves this by making the reference matrix for A_j the same as that for A_i. The data transformation matrix $t_j = f_i f_j^{-1}$ applied to array A_j modifies the reference $A_j(f_j(\vec{I}))$ to be $A_j(t_j f_j(\vec{I}))$ which is the same as $A_j(f_i(\vec{I}))$, so both lhs data elements are co-located.

Computation Alignment can achieve the same objective in an analogous fashion. First assume that there is no pre-existing data alignment between arrays A_i and A_j, that is $A_i(r(\vec{I}))$ and $A_j(r(\vec{I}))$ are co-located, for some reference matrix r. Instead of aligning the arrays, statement S_j can be computationally aligned to statement S_i by transforming S_j using matrix $T_j = f_i^{-1} f_j$, which

[4] Only some of these alignments may be legal.

Algorithm: *B2*
input: K statements $\ni_{owner\ (A_i(f_i(\vec{I})))}(S_i)$, for $i = 1..K$.
output: Alignment transformation T_i for each S_i

```
begin
1.  if statements are mapped onto a single processor then return // Since, there is no need for alignment.

// Search for legal alignments with a single processor
2.  for i = 1, K
        α_i ← ∅
        for j = 1, K
            // add to α_i  transformation to align S_j to S_i
            α_i ← α_i ∪ {T_j}, where T_j = f_i^{-1} d_j f_j, assuming A_j is data aligned to A_i by d_j.
        end for
    end for
    A_all ← {α_1, ..., α_K} ∪ {I, ..., I}

// first candidate alignment is the identity alignment
3.  α ← {I, ..., I}
4.  for i = 1, K
        if α_i ∈ A_all is legal and has lower communication than α then
            α ← α_i
        end if
    end for
5.  if α ≠ {I, ..., I} then return // found a legal alignment

// The only legal alignment with one processor is the identity or legal alignments
// have higher communication than the identity alignment.
// Therefore, search for legal alignments with fewest possible processors
6.  S ← {S_i  |  1 ≤ i ≤ K}
7.  while S is not empty
        for j = 1, m
            g_j ← {S_i  |  1 ≤ i ≤ K and S_i  mapped onto processor  j}
        end for
8.      G ← {g_i  |  1 ≤ i ≤ m}
9.      g ← largest set in G
10.     S ← S − g
11.     order statements in S (which are not in g) in increasing size of sets of G they are in.
12.     for all statements S_i of S in the order
            T_i ← transformation to align S_i to a statement in g
            if T_i is legal then
                // T_i is the selected transformation for S_i
                g ← g ∪ {S_i}
                S ← S − {S_i}
            end if
        end for
    end while
end
```

Figure 6.4: Algorithm *B2* to derive Computation Alignment.

ensures that *owner* $(A_i(f_i(\vec{I})))$ and *owner* $(A_j(f_j'(\vec{I})))$ both refer to the same processor, when f_j' is the new reference matrix after applying T_j. The transformation matrix T_j is derived by solving $f_j T_j^{-1} = f_i$ for T_j. First, both sides of the equation by are pre-multiplied f_j^{-1}, and then, both sides of the equation are inverted. A pre-existing data alignment between arrays A_i and A_j, say d_j, can be accounted for by computationally aligning S_j to S_i by transformation matrix $f_i^{-1} d_j f_j$.

Given the computation alignments in $\mathcal{A}_{all}$, we search for those that are legal in Steps 3–5, and choose the one with lower communication overhead. As our initial point of comparison, we choose the identity transformation, where the statements are not transformed at all (Step 3). In Step 4, we iterate through the sets of alignment in $\mathcal{A}_{all}$ and select alignments α which results in a legally transformed loop with lower communication overhead.

If none of the alignments in $\mathcal{A}_{all}$ are legal, then Steps 6–12 derive alignments that map the statements onto (more than one, but) as few processors as dependences permit. In Step 6, S is initialized to be set of all K statements in the loop body. In each iteration of Step 7, we remove from S the statements that can be legally mapped onto the same processor. The statements in S are organized into sets g_i, where statements in a set g_i are all mapped onto the same processor i; G of Step 8 is the set of these sets g_i. In Steps 9 and 10, we select the largest set g in G and remove the statements of g from S. The objective of Steps 11 and 12 is to add to g as many statements from remaining statements in S as possible. In Step 11, we order the statements in S according to the size of the set g_i they are in, starting with the statements in the smallest sets. In Step 12, we attempt to align the statements in S to the statements in g in the order specified. A statement S_i that can be legally aligned to the statements in g is aligned, added to g, and removed from the set S. The rationale behind the ordering in Step 11 is that the probability is higher that *all* statements of a smaller set can be aligned to g, resulting in a smaller number of sets in G.

As an example of applying algorithm $B2$, the loop on the top of Figure 6.5 is transformed into the loop at the bottom. The only candidate alignments are aligning $S1.1$ to $S1.2$ or aligning $S1.2$ to $S1.1$. Both alignments are equivalent in that they are inverses of each other. The transformed loop shown has $S1.1$ aligned to $S1.2$. Note that, both $S1.1$ and $S1.2$ are now mapped onto the same processor in every iteration. Note that the guards can be removed from the transformed loop with the techniques discussed in Section 4.2.

```
for i = 2, n
  for j = 2, n
    S1.1 : ⊙A(i-1,j) [ t(i,j) = A(i-1,j) + B(i-1,j) + C(i-1,j)+
                A(i-1,j-1) + B(i-1,j-1) + C(i-1,j-1) ]
    S1.2 : ⊙A(i,j) [ A(i,j) = t(i,j) + A(i,j-1) + B(i,j-1) + C(i,j-1) ]
  end for
end for

          for i = 1, n
            for j = 2, n
              S1.2 : (i > 1)  ⊙A(i,j)  [ A(i,j) = t(i,j) + A(i,j-1) + B(i,j-1) + C(i,j-1) ]
              S1.1 : (i < n)  ⊙A(i,j)  [ t(i+1,j) = A(i,j) + B(i,j) + C(i,j)+
                    A(i,j-1)  +  B(i,j-1)+C(i,j-1)  ]
            end for
          end for
```

Figure 6.5: Computation Alignment of a loop with ⊙ operators.

6.3.3 Data Alignment of Temporary Arrays

Algorithm *B3* of Figure 6.6 is applied after algorithm *B2* in order to properly data align the temporary arrays that may have been introduced by algorithm *B1*. *B3* data aligns each temporary array to the array used in the ⊙ operator for the respective statement. If T_i is the alignment transformation that algorithm *B2* applied to statement S_i, $t_i(T_i^{-1}(\vec{I}))$ is the lhs of the transformed S_i, and $⊙_{A_i(f_i T_i^{-1}(\vec{I}))}$ is the operator that maps transformed S_i onto a processor, then algorithm *B3* data aligns t_i to array A_i by transformation matrix f_i. As a result of this alignment, $t_i(T_i^{-1}(\vec{I}))$ becomes $t_i(f_i T_i^{-1}(\vec{I}))$. Because the reference to A_i in the ⊙ operator and to the t_i on the lhs are now the same, the A_i reference in the ⊙ operator can be replaced by t_i. In the example loop of Figure 6.5, t is aligned to A such that $t(i+1,j)$ becomes $t(i,j)$ and $t(i,j)$ becomes $t(i-1,j)$.

6.3.4 Summary of Stages to Pack Iterations

The three stages in packing iterations can be summarized as follows:

(i) Algorithm *B1* decomposes the statements so that each statement of the decomposed loop body has a single ⊙ operator for the entire statement.

(ii) Algorithm *B2* aligns the statements of the decomposed loop body so that they are mapped onto as few processors as possible.

(iii) Finally, algorithm *B3* data aligns temporary arrays on the lhs of the decomposed statements to the arrays used in the respective ⊙ operators.

Algorithm : *B3*
input: Alignment $\alpha_k = \{T_1, \ldots, T_K\}$.
output: Data alignment for temporary arrays introduced in Stage 1.
begin
    for each statement $S_i : \odot_{A_i,(f_i(\bar{I})T_i^{-1})}(S_i)$ with lhs t_i
        data align t_i to A_i by f_i
    end for
end

Figure 6.6: Algorithm *B3* to data align temporary arrays.

As a result of applying these stages, any loop with (possibly multiple) $\odot$ operators at the subexpression granularity is converted to an equivalent loop for which the familiar owner-computes rule is to be applied.

6.4 Scanning Local Iteration Space

In this section, we show how to derive new loop bounds so that the processors execute only their local iteration space. We adapt the guard elimination techniques described in Section 4.2 for this purpose. The techniques described here can be viewed as a generalization of existing techniques that peel iterations off the local iteration space in order to isolate the iterations where all statements are mapped onto the same processor [21].

The local computation space of a statement for a given processor is a subset of the total computation space for the statement. The extent of the local computation space for each processor is determined by the ownership test for the statement; the computations are executed only in those iterations where the ownership test for the statement evaluates to true.[5] The local iteration space of the loop body is the union of the local computation spaces of all the statements of the loop body projected onto a grid that has the same dimension as the loop.

We distinguish between three types of loop bodies — a loop body with a single statement, a loop body with multiple statements which are all mapped onto the same processor, and finally, a loop body with multiple statements which may be mapped onto more than one processor.

First, consider the derivation of the loop bounds for the local iteration space of a loop body with only one statement. Obviously, the local computation space of the statement is also the local

[5] At this stage, after having applied *B1*, *B2* and *B3*, there exists a single $\odot$ operator in every statement.

iteration space of the loop body. When the computation rule for the statement is specified indirectly using an array A, then the bounds of the local iteration space are obtained by analyzing the local data space of A, the set of elements of A mapped locally onto the processor.[6] We first represent the local data space of the array with a set of inequalities, and use this set and guard elimination techniques to derive a set of inequalities representing the local iteration space.

To see how to represent the data space of an array as a set of inequalities, consider a 2-dimensional array distributed by rows. If $\vec{I_d} = (i_1, i_2)^t$ is the vector that indexes elements of the array, then the inequalities $l \leq i_1 \leq u$ and $1 \leq i_2 \leq n$ represent a block of rows, where l and u provide bounds for the block of data on a processor.[7] For the loop on the extreme right of Figure 6.1 on page 83, l and u for both arrays A and B were $p * b$ and $(p+1) * b - 1$, respectively. The inequalities for the local data space for A can be represented by matrix-vector notations, say $\beta_d \vec{I_d} \geq \vec{0}$, similar to the specification of an iteration spaces.

For a statement S : $\odot_{A(f(\vec{I}))}$ ... the local iteration space can be derived from A's data space, $\beta_d \vec{I_d} \geq \vec{0}$, by replacing $\vec{I_d}$ by $f\vec{I}$ to become $\beta_d f \vec{I} \geq \vec{0}$, since the reference matrix f provides indexes to array elements. Now, inequalities $\beta_d f \vec{I} \geq \vec{0}$ are similar to a guard for the statement. Therefore, the new loop bounds can be obtained by applying the guard elimination techniques of Section 4.2, where the statement has guard of $\beta_d f \vec{I} \geq \vec{0}$ and original loop bounds are specified by, say $\beta \vec{I} \geq \vec{0}$.

Now consider the case where a loop body contains multiple statements mapped onto the same processor. The method outlined above for a single statement loop body is also applicable in this case, since the local iteration space of any of the statements in the loop body is also the local iteration space of the loop body.

Finally, consider a loop body with multiple statements, which are not all mapped onto the same processor. In this case, the loop bounds should be modified so that the processor iterates through the *union* of local iteration spaces for the statements. The bounds of the local iteration space for each statement are obtained by analyzing the local data spaces as we did for a loop body with a single statement. The inequalities describing the local iteration space of a statement serves as its guards. The guard elimination techniques can then be used to isolate iterations where all statements are mapped onto the same processor.

As an example of deriving new loop bounds for a local iteration space, consider the nested loop on the top of Figure 6.7, which is the core of the CDA transformed loop at the bottom of

[6] The arrays are typically mapped onto processors using some basic distribution strategy such as distribution by rows, columns, blocks etc.

[7] Cyclical mapping functions are characterized by non-unit strides.

```
for i = 2, n - 1
  for j = 2, n
    S1.2 :  ∃A(i,j) [ A(i,j) = t(i - 1,j) + A(i,j - 1) + B(i,j - 1) + C(i,j - 1) ]
    S1.1 :  ∃t(i,j) [ t(i,j) = A(i,j) + B(i,j) + C(i,j) + A(i,j - 1) + B(i,j - 1) + C(i,j - 1) ]
  end for
end for

        for i = 2, n - 1
          for j = 2, n
            S1.2 : (l_i ≤ i ≤ u_i) ∧ (l_j ≤ j ≤ u_j)
                  A(i,j) = t(i - 1,j) + A(i,j - 1) + B(i,j - 1) + C(i,j - 1)
            S1.1 : (l_i ≤ i ≤ u_i) ∧ (l_j ≤ j ≤ u_j)
                  t(i,j) = A(i,j) + B(i,j) + C(i,j) + A(i,j - 1) + B(i,j - 1) + C(i,j - 1)
          end for
        end for

for i = l_i, u_i
  for j = l_j, u_j
    S1.2 :  A(i,j) = t(i - 1,j) + A(i,j - 1) + B(i,j - 1) + C(i,j - 1)
    S1.1 :  t(i,j) = A(i,j) + B(i,j) + C(i,j) + A(i,j - 1) + B(i,j - 1) + C(i,j - 1)
  end for
end for
```

Figure 6.7: Modification of loop bounds to scan local iteration space.

Figure 6.5 after eliminating guards. Let the arrays accessed in the loop be distributed along both array dimensions such that a processor's data space is $l_i \leq i_1 \leq u_i$ and $l_j \leq i_2 \leq u_2$, where the l's and u's are functions of array size and a number identifying a processor. In this case, the inequalities for the local data space also serve as the inequalities for the local iteration space, since the reference matrices for both $A(i,j)$ and $t(i,j)$ are the identity matrix. These inequalities are used to guard the statements of the loop as shown in the center of the figure. Eliminating these guards as described in Section 4.2 leads to the new loop bounds that scan the local iteration space as shown at the bottom of the figure.[8]

[8]The local data space may not always define the bounds for all the iterators. This happens in cases where the array reference used to derive the data space is a function of only some of the iterators, and in cases where the loop has a dimension different from the array dimension. In these cases, instead of taking the bounds for the iterator from the bounds of the local data space, they are taken from the bounds of the loop iteration space.

CHAPTER 7

Other Applications of CDA

Ill can he rule the great that cannot reach the small.
— *Edmund Spenser: Book v. Canto ii. St. 43*

In this chapter we describe five additional applications of CDAs to:

1. Improve instruction level parallelism.

2. Eliminate synchronizations.

3. Generalize loop distribution.

4. Transform imperfect loop nests, and

5. Improve global optimizations.

We primarily use examples to illustrate these applications of CDA.

7.1 Improving Instruction Level Parallelism

Most processor architectures support instruction level parallelism, say in the form of instruction pipelines and multiple functional units. A compiler targeting these processors applies techniques that re-order instructions so that this parallelism can be better exploited in order to improve processor efficiency. These techniques often attempt to group unrelated instructions (i.e. those having no dependences between them) so that their execution can be overlapped. *Software pipelining* is one such compiler technique: it interleaves instructions from adjacent loop iterations [30]. CDA can also be used to interleave computations from multiple iterations, and can thus be viewed as a generalization of the software pipelining principle. In the case of CDA, the objective is to modify the constitution of iterations to have as few dependences as possible, thus allowing instruction

```
for j = 2, n
  for i = 2, n
    S : A(i,j) = A(i-1,j) + B(i-1,j) + C(i-1,j) + A(i,j-1)+
          B(i,j-1) + C(i,j-1) + A(i-1,j-1) + B(i-1,j-1) + C(i-1,j-1)
  end for
end for
```

```
          ⋮
        for j = 2, n-1
          for i = 2, n
  ⟹         S1 : A(i,j) = A(i-1,j) + B(i-1,j) + C(i-1,j) + A(i,j-1) + t(i,j)
            S2 : t(i,j+1) = B(i,j) + C(i,j) + A(i-1,j) + B(i-1,j) + C(i-1,j)
          end for
        end for
          ⋮
```

Figure 7.1: Application of CDA transformation to improve instruction level parallelism.

scheduler more freedom to schedule instructions. Thus, CDA can be used as a high order transformation to strengthen existing instruction scheduling techniques. Nevertheless, CDA differs from instruction scheduling techniques in a number of fundamental ways:

(i) CDA is a higher order transformation than instruction scheduling — because CDA is a source level transformation the granularity of the computations being considered in the CDA framework is usually larger than the granularity of single instructions considered by the instruction scheduling techniques.

(ii) CDA can move instructions across iterations in any loop dimension, whereas instruction schedulers move instructions within a basic block or across adjacent iterations of the innermost loop.

(iii) CDA tends to be used to target higher level optimization objectives than instruction level parallelism.

(iv) CDA introduces overheads that arise from modification to the loop structure, including the need for guard computations and storage requirements for the temporary variables. In contrast, instruction scheduling techniques have much lower overhead. For example, the overheads of software pipelining are limited to the loop overhead in prolog and epilog loops.

As an illustration of applying CDA to improve instruction level parallelism, consider the loop at the top of Figure 7.1. The left to right execution semantics in Fortran results in a linear chain of dependent arithmetic operations for statement S. That is, the instruction sequence generated

for the rhs of statement S is such that each instruction in the sequence is dependent on the result of the previous instruction in the sequence. Instructions in such linear chains cannot be easily re-ordered, since the dependences prevent the movement of instructions for interleaved execution. If, in addition, the instructions require multiple cycles to complete then the pipeline cannot be kept full. For this reason, instruction schedulers attempt to fill the pipeline with other instructions that do use the result. Unfortunately, in the case of the loop considered, dependences prevent the interleaving of any of the addition operations. Similarly, on a processor with multiple functional units, the execution chain only allows the employment of only one unit at a time.

An appropriate CDA transformation can help in this case. For example, the equivalent CDA transformed loop at the bottom of Figure 7.1 leaves open much more opportunity for instruction scheduling. The additions are now arranged in two linear chains, one for each of statements S_1 and S_2. The additions in the two chains originally belonged to two different iterations, namely j and $j+1$. Although there are dependences between additions in a chain, there are no dependences between the two chains. Therefore, the execution of instructions in the two chains can be interleaved, while preserving the left to right Fortran execution semantics of the original loop.[1]

7.2 Eliminating Synchronizations

Dependences in a loop can limit the availability of coarse-grain parallelism. When the rank of the dependence matrix is not less than the loop dimension, then the loop nest does not have any parallel outer loops (although all the inner loops can be made parallel) [29]. Unfortunately, the performance of parallel inner loops is not scalable, since the dependences in the outer loops manifest themselves as barrier synchronizations. CDA transformations can be used to modify the dependence matrix of some loops, so that the rank becomes less than the loop dimension; this effectively eliminates the need for synchronization and thus introduce the availability of coarse-grain parallelism.

In this section, we show how to derive CDA transformations that eliminate synchronization by modifying loop carried dependences between statements into loop independent dependences. The technique is general in that it can be used to modify a loop carried dependence $\vec{d}$ between two statements S_w and S_r into any desired dependence $\vec{d'}$, although in this case we would modify $\vec{d'}$ to be $\vec{0}$. Let A_w and A_r be the reference matrices of the write reference w to an array in statement S_w

[1] Such opportunities to interleave operations are also essential to pack the instructions of very large instruction word (VLIW) machines with useful computations.

and a read reference r to the array in statement S_r, respectively.[2] We can derive a matrix f, which transforms the computations of S_r so that the dependence between w and the modified reference r' becomes $\vec{d'}$, as follows. Let A_w, A_r and f^{-1} be:

$$A_w = \begin{bmatrix} U_w & \vec{0} \\ 0 & 1 \end{bmatrix} \quad A_r = \begin{bmatrix} U_r & \vec{d_r} \\ 0 & 1 \end{bmatrix} \quad f^{-1} = \begin{bmatrix} T & \vec{t} \\ 0 & 1 \end{bmatrix}$$

Transforming the computations of S_r by matrix f modifies the . reference matrix A_r to become $A_r f^{-1}$. If the dependence between statements S_w and S_r in the original loop was $\vec{d}$, then reference matrices $A_w \vec{I}$ and $A_r \vec{I} + A_r \vec{d}$ both access the same array element in the original loop. We would like $A_w \vec{I}$ and $A_r \vec{I} + A_r \vec{d'}$ to access the same array element after the transformation; i.e.

$$A_w \vec{I} = A_r f^{-1}(\vec{I} + \vec{d'})$$

By substituting for A_w, A_r and f^{-1} in this equation, we obtain:

$$\begin{bmatrix} U_w & \vec{0} \\ 0 & 1 \end{bmatrix} \vec{I} = \begin{bmatrix} U_r T & U_r \vec{t} + \vec{d_r} \\ 0 & 1 \end{bmatrix} (\vec{I} + \vec{d'})$$

After expanding the matrix-vector multiplication on the right hand side we have:

$$U_w \vec{I} = U_r T(\vec{I} + \vec{d'}) + U_r \vec{t} + \vec{d_r}$$

which can be expanded into:

$$U_w \vec{I} = U_r T \vec{I} + U_r T \vec{d'} + U_r \vec{t} + \vec{d_r}.$$

For this equation to be true for all iterations $\vec{I}$, it is necessary for T to be equal to $U_r^{-1} U_w$, and for $U_r \vec{t}$ to be equal to $-U_r T \vec{d'} - \vec{d_r}$. By substituting $T = U_r^{-1} U_w$ in this latter equation, we obtain:

$$U_r \vec{t} = -U_w \vec{d'} - \vec{d_r}$$

[2]The matrices are suitably padded when the array and loop dimensions are not the same.

```
for i = 1,n                                 forall i = 0,n
  for j = 1,n                                 for j = 1,n
    S1 : A(i,j) = A(i,j - 1)                    (i > 0) S1 : A(i,j) = A(i,j - 1)
    S2 : B(i - 1,j) = A(i - 1,j)       ==>      (i < n) S2 : B(i,j) = A(i,j)
  end for                                     end for
end for                                     end for
```

Figure 7.2: CDA transformation to eliminate barrier synchronization.

which we can solve for $\vec{t}$ by pre-multiplying both sides of the equation by U_r^{-1}:

$$\vec{t} = -U_r^{-1}U_w\vec{d'} - U_r^{-1}\vec{d_r}$$

Therefore, matrix f is defined such that:

$$f^{-1} = \begin{bmatrix} U_r^{-1}U_w & -U_r^{-1}U_w\vec{d'} - U_r^{-1}\vec{d_r} \\ 0 & 1 \end{bmatrix}$$

which modifies dependence $\vec{d}$ between statements S_w and S_r to be $\vec{d'}$. In particular, when the desired dependence $\vec{d'}$ is $\vec{0}$, then the transformation f is defined such that:

$$f^{-1} = \begin{bmatrix} U_r^{-1}U_w & -U_r^{-1}\vec{d_r} \\ 0 & 1 \end{bmatrix}$$

In order to illustrate such an application of CDA to expose outer loop parallelism, consider the loop on the left hand side of Figure 7.2. This loop does not have any outer loop parallelism because of the $(1,0)$ and $(0,1)$ dependences. That is, iterations (i,j) must wait until iterations $(i-1,j)$ and $(i,j-1)$ have completed. Because the loop has only inner loop parallelism, barrier synchronization between outer loop iterations is necessary. The $(1,0)$ dependence can be eliminated by linearly transforming the S_2 computation space relative to the S_1 computation space, defining transformation matrix f such that T is the identity matrix and t is $(1,0)^t$. The transformed loop is shown on the right hand side of Figure 7.2. The transformation modifies the $A(i-1,j)$ reference in S_2 to $A(i,j)$, so that the transformed loop now only has a $(0,1)$ dependence. Thus the new loop nest has an outer loop that is fully parallelizable.

The technique we just described is a generalization of loop alignment [2]. While loop alignment considers only offset alignments between statements, CDA transformations can be any non-singular integer matrices. Moreover, our generalization can align subexpressions, statements or subnests in

```
for i = 1,n                                        for i = 1,n
  S1 : A(i) = x                                      S1 : A(i) = x
  S2 : C(i) = A(i - 1) + D(i - 1)                  end for
  S3 : D(i) = C(i)                      ==>        for i = 1,n
end for                                              S2 : C(i) = A(i - 1) + D(i - 1)
                                                     S3 : D(i) = C(i)
                                                   end for
```

Figure 7.3: An example loop distribution.

the loop body.[3] Thus, CDA can be viewed as unifying loop alignment and its generalization into a linear algebraic framework.

7.3 Generalizing Loop Distribution

Loop distribution divides a loop body into groups of statements and creates a separate nested loop for each group [56, 58]. In this section, we show how CDA includes loop distribution and its generalization into the linear algebraic framework.

As an example of loop distribution, consider the loop on the left hand side of Figure 7.3. This loop is distributed to create two loops, the first with statement S_1 and the second loop with statements S_2 and S_3.

Loop distribution can be effective in improving parallelism, since the new loop nests contain only a subset of the dependences in the original loop.[4] However, loop distribution is restricted in that a loop distribution is legal only if it keeps the statements participating in a dependence cycle in the same loop nest. For instance, statements S_2 and S_3 of loop on the left hand side of Figure 7.3 participate in a dependence cycle, so they must belong in the same loop. Hence, some nested loops such as the loop shown in Figure 7.4 cannot be distributed, since all of the statements in the loop body participate in a dependence cycle.

The CDA framework generalizes loop distribution in three ways:

- First, any loop distribution can be represented by a CDA transformation that decomposes the loop body and applies appropriate offset alignment along the outermost loop dimension.

[3] When the dependences are between references of the same statement, then the rank of the dependence matrix cannot be changed, because the effect of eliminating one dependence is offset by the effect of adding a new dependence on references to the temporary array. In this case, a CDA transformation can only modify the structure of dependences.

[4] Loop distribution is also useful in improving cache utilization and reducing register pressure by distributing array accesses in the loop body into different loop nests.

```
for i = 1.n
   S1 : D(i) = A(i - 1) + A(i + 1) + A(i + 2)
   S2 : A(i) = B(i) + C(i)
end for
```

Figure 7.4: A loop that cannot be distributed.

```
for i = 1.n
   S1 : A(i) = x
   S2 : C(i) = A(i - 1) + D(i - 1)          ==>
   S3 : D(i) = C(i)
end for

for i = 1.2 * n
   S1 : (1 <= i <= n)           A(i) = x
   S2 : (n + 1 <= i <= 2 * n)   C(i - n) = A(i - n - 1) + D(i - n - 1)     ==>
   S3 : (n + 1 <= i <= 2 * n)   D(i - n) = C(i - n)
end for

for i = 1.n
   S1 : A(i) = x
end for
for i = n + 1.2 * n
   S2 : C(i - n) = A(i - n - 1) + D(i - n - 1)
   S3 : D(i - n) = C(i - n)
end for
```

Figure 7.5: Loop distribution as a CDA transformation.

to each group of statements (to be distributed) so that the computation space for each group does not overlap with the computation space for any other group. The first group is aligned by an offset of 0, whereas the ith group is aligned by an offset of $ni - n$, where n is the of size of the outermost loop. For instance, the loop distribution of Figure 7.3 can be effected by the CDA transformation that aligns statements S_2 and S_3 by an offset of n. The transformed loop with and without guards is shown in the center and right hand side of Figure 7.5. Note, however, that the loops resulting from loop distribution and the loops resulting from the CDA transformation differ in subscript functions.

- Second, with CDA, loop distribution can be performed at the granularity of subexpressions, and not just entire statements. Computation decomposition can store the results of subexpressions in temporaries to isolate dependence cycles. Thus, appropriate computation decomposition can be considered as a form of *node splitting*, which introduces temporaries to

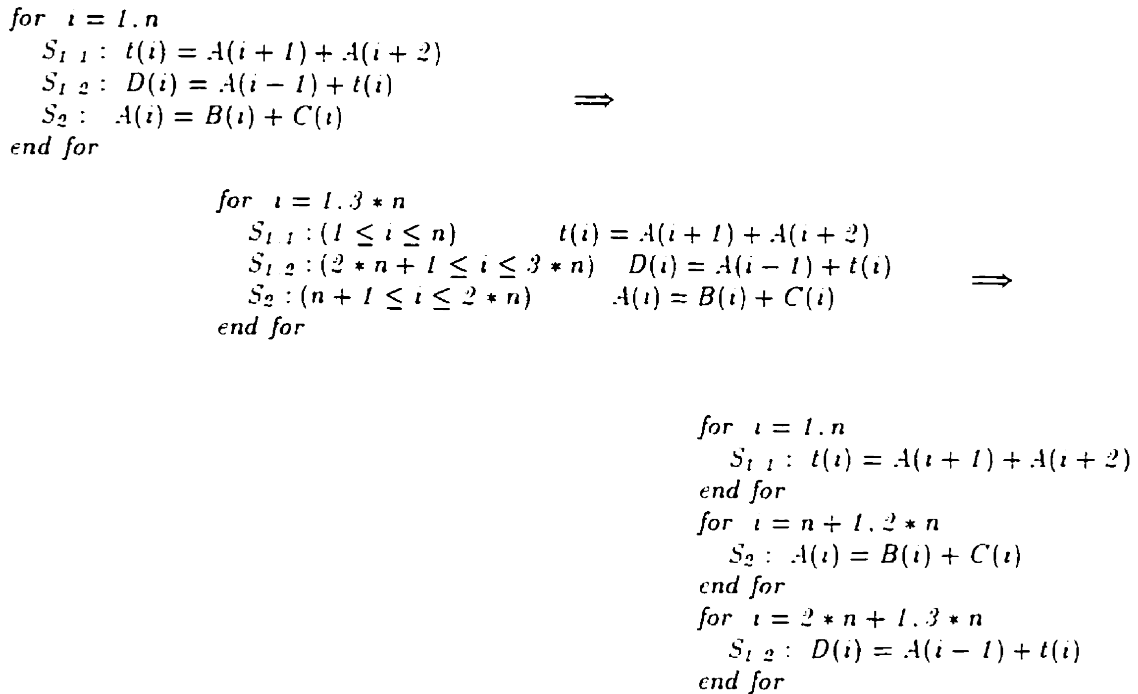


Figure 7.6: An example of breaking dependence cycles to enable loop distribution.

break dependence cycles [58]. For example, statement S_1 in the loop of Figure 7.4 can decomposed into statements $S_{1.1}$ and $S_{1.2}$ as shown on the left hand side of Figure 7.6. The decomposed loop does not have a dependence cycle, so it can now be distributed. The center of Figure 7.6 shows how the statements can be aligned to effect one possible loop distribution. The net effect of this transformation is improved parallelism: the original loop could not be parallelized, whereas all three loops shown on the right hand side of Figure 7.6 can be.

- Third, CDA makes *partial* loop distributions possible. A loop distribution separates *all* instances of a statement from the instances of another statement in the loop body. A partial loop distribution separates only *some* instances of a statement or subexpression from the instances of other statements or subexpressions in the loop body. A partial loop distribution would be beneficial in a situation where a dependence cycle prevents loop distribution (and where node splitting does not help break the dependence cycle).

  As an example, consider the loop on the left hand side of Figure 7.7 which has a dependence cycle between statements S_1 and S_2. The dependence cycle cannot be isolated with computation decomposition, since the dependences in the cycle are flow dependences.[5] However, S_2

[5] That is, all statements of the decomposed loop will be in the dependence cycle.

```
L:  for  i = k + 1, n
      for  j = 1, n
        S1 :  B(i,j) = A(i - 1,j)
        S2 :  A(i,j) = B(i - k,j)
      end for
    end for
```

$\Longrightarrow$

```
L:  for  i = 1, n
      for  j = 1, 2 * n
        S1 :  (k + 1 ≤ i ≤ n)   B(i,j) = A(i - 1,j)
        S2 :  (1 ≤ j ≤ n - k)   A(i + k,j) = B(i,j)
      end for
    end for
```

$\Longrightarrow$

```
L1 :  for  i = 1, k
        for  j = 1, n
          S1 :  A(i + k,j) = B(i,j)
        end for
      end for
L2 :  for  i = k + 1, n - k
        for  j = 1, n
          S2 :  B(i,j) = A(i - 1,j)
          S1 :  A(i + k,j) = B(i,j)
        end for
      end for
L3 :  for  i = n - k + 1, n
        for  j = 1, n
          S2 :  B(i,j) = A(i - 1,j)
        end for
      end for
```

Figure 7.7: CDA transformation for partial loop distribution.

can be aligned by an offset of $-k$ to obtain the transformed loop shown in the center of the figure. The CDA transformed loop after eliminating the guards is shown on the right hand side of Figure 7.7. The transformation effected a partial loop distribution where only k computations (out of n) of the statements S_1 and S_2 are separated from each other.[6] When the dependent iterations are far apart (i.e. k is large), then partial loop distribution can separate a substantial number of statement instances.

As another example, consider the loop on the left hand side of Figure 7.8 which cannot be distributed due to a dependence cycle that cannot be broken. CDA can align S_2 by an offset along the j dimension in this case so that the statements are distributed only with respect to j and k dimensions. The CDA transformed loops with and without guards are shown at the center and right of Figure 7.8.

[6] The dependence cycle prevents the distributions of the computations of the statements in the remaining iterations.

```
L :  for  i = 1,n
       for  j = 1,n
         for  k = 1,n
           S1 :  B(i,j,k) = A(i - 1,j,k)
           S2 :  A(i,j,k) = B(i - 1,j,k)                ==>
         end for
       end for
     end for

     L :  for  i = 1,n
            for  j = 1,n
              for  k = 1,n
                S1 :  (1 <= j <= n)  B(i,j,k) = A(i - 1,j,k)
                S2 :  (n + 1 <= i <= 2 * n)  A(i,j - n,k) = B(i - 1,j - n,k)      ==>
              end for
            end for
          end for

          L :  for  i = 1,n
                 L1 :  for  j = 1,n
                         for  k = 1,n
                           S1 :  B(i,j,k) = A(i - 1,j,k)
                         end for
                 end for
                 L2 :  for  j = n + 1,2 * n
                         for  k = 1,n
                           S2 :  A(i,j - n,k) = B(i - 1,j - n,k)
                         end for
                 end for
               end for
```

Figure 7.8: CDA transformation for partial loop distribution.

- Finally, any sequence of loop distribution, linear transformation of the new loop nests, and fusion of the transformed loop nests can be represented by a single CDA transformation.[7] A linear transformation applied to a loop nest after loop distribution is essentially an alignment transformation applied to the computation space of the corresponding statement(s). The loop fusion is effected while generating code for the CDA transformed loop, where the computation spaces are coalesced by projecting them onto a grid and finding their union.

7.4 Transforming Imperfect Loop Nests

The model of nested loops, described in Chapter 3, includes both perfectly nested and imperfectly nested loops. Both linear loop and CDA transformations can be applied to perfectly nested loops.

[7] Loop fusion coalesces adjacent loop nests into a single loop nest [58].

```
for i = 1,n                                  for i = 1,n
  S1 : A(i,0) = C                              for j = 1,n
  for j = 1,n                                    (j = 1) S1 : A(i,0) = C
    S2 : A(i,j) = A(i,j-1) + D     ==>                   S2 : A(i,j) = A(i,j-1) + D
  end for                                      end for
end for                                      end for
```

Figure 7.9: Converting a simple imperfectly nested loop into a perfectly nested loop.

```
for i = 0,n                                  for i = 0,2n+1
  for j = 0,n                                  for j = max(0,i-n-1),min(n,i)
    S1 : A(i,j) = A(i-1,j) + T1                  S1 : (i <= 2n, max(0,i-n) <= j <= min(i,n))
  end for                                                A(j,i-j) = A(j-1,i-j) + T1
  for j = 0,n                      ==>           S2 : (i >= 1, max(0,i-n-1) <= j <= min(i-1,n))
    S2 : A(i,j) = A(i,j+1) + T2                          A(j,i-j-1) = A(j,i-j) + T2
  end for                                      end for
end for                                      end for
```

Figure 7.10: CDA transformation of an imperfectly nested loop.

Although transformation of unconstrained imperfectly nested loops is still an open issue, we show in this section how some imperfectly nested loops can be successfully transformed using linear loop and CDA transformations.

Simple cases of imperfect nests occur, for example, in order to perform boundary computations or initializations. The cause of imperfectness is often a single assignment statement interspersed between the loop statements of an otherwise perfectly nested loop. In this case, the imperfect nest can be transformed into a perfect nest by moving the single assignment statement into the loop and using guards [1, 56]. An example of this is shown in Figure 7.9. After this transformation, the nested loop can be applied a linear loop transformation.

Another frequently occuring imperfectly nested loop structure, where the loop body of a perfectly nested loop consists of a sequence of perfect subnests is of particular interest to us here. The left hand side of Figure 7.10 shows such an imperfect loop nest. Such nested loops can be linearly transformed only in a hierarchical fashion — each of the perfect subnests can be linearly transformed, and the loop nest containing the sequence of subnests can be linearly transformed.[8] However, the entire loop nest cannot be transformed by a linear transformation.

With CDA, it is possible to transform this type of imperfect loop nest. For an example, the imperfect nest on the left hand side of Figure 7.10 can be CDA transformed to the loop nest on the right. CDA treats all computations of S_1 (in the i and j loops) as one computation space,

[8] In particular, the loop on the left hand side of Figure 7.10 can be linearly transformed as two 1-dimensional loops with j as their iterator, and as a 1-dimesional loop with i as its iterator.

```
L1 : for i = 1,n                              L1 : for i = 1,n
       for j = 1,n                                   for j = 1,n
         A(i,j) = A(i,j - 1) + C                       A(i,j) = A(i,j - 1) + C
       end for                                       end for
     end for                                       end for

L2 : for i = 1,n                   ==>        L2' : for i = 0,n
       for j = 1,n                                   for j = 1,n
         S1 : A(i,j) = A(i,j - 1) + D                  S1 : (i > 0)  A(i,j) = A(i,j - 1) + D
         S2 : B(i,j) = A(i - 1,j - 1) + E              S2 : (i < n)  B(i + 1,j) = A(i,j - 1) + E
       end for                                       end for
     end for                                       end for
```

Figure 7.11: Effect of CDA on global optimization.

while all computations of S_2 in i and j loops are treated as the second computation space. The transformation first aligns the computation space of S_2 by an offset of -1 along the j dimension; and then skews both computation spaces so that the dependences are internalized to the inner loop. In the transformed loop, the flow dependence from S_1 to S_2 is loop independent, and the flow dependence from S_2 to S_1 is carried only by the j iterations. Therefore, the iterations of the outer loop are independent, so that the outer loop can be executed in parallel. Transformations of this type can expose additional optimization opportunities that linear loop transformations alone cannot.[9]

7.5 Using CDA to Improve Global Optimization

CDA can also be used for improving *global optimization* [4, 19, 34], since certain CDA transformations are duals of certain data transformations. For example, in Chapters 5 and 6, we described how CDA transformations can be used to achieve the same effect as array padding and data alignment. The advantage of using CDA instead of making global changes to data is that CDA only changes the local loop structure and thus reduces the number of constraints on a global optimization algorithm.

CDA can also be used to modify a loop structure to suit the data partitioning imposed by adjacent loops. For example, consider loop L_1 on the left hand side of Figure 7.11 alone. When array A is distributed onto processors by rows, then this loop does not require any communications, since the (parallel) outer loop can then be mapped onto processors so that each processor accesses the rows of A locally. However, if we consider both loops L_1 and L_2 together, then a row-wise

[9] A linear loop transformation cannot be applied to internalize the dependence and expose parallelism in the outer loop, because the loop is imperfectly nested. Linearly transforming the i loop or the j loops alone cannot not expose parallelism either, since the dependences are across j subnests.

mapping of A results in communication in L_2: This is because linear loop transformations of L_2 can only expose parallelism in the inner loop, and communications are necessary. It is possible, however, to eliminate communication by CDA transforming L_2 to suit the data partitions of A, chosen according to L_1. Communication arises in L_2 because of the $(1, 1)$ dependence between statements S_1 and S_2. This dependence can be modified to be a $(0, 1)$ dependence by aligning S_2 to S_1. The alignment shifts the S_2 computations relative to the S_1 computations along the i iterator of the L_2 loop. The transformed loop L_2' is shown on the right hand side of the figure, which has a parallel outer loop and does not require communication with a row-wise distribution of A.

7.6 Summary

This chapter has shown that there are many uses for the CDA transformation framework that go beyond what existing techniques achieve. In particular, we showed how some of the existing loop transformation techniques and their generalizations can be unified into the linear algebraic framework of CDA. However, CDA needs to be studied further with respect to:

- the heuristics to derive CDA transformations for the optimizations mentioned in the chapter,
- the extent to which CDA can be effective, and
- integration of CDA with techniques for which CDA is a dual.

CHAPTER 8

Application of CDA to Example Nested Loops

When the way comes to an end, then change - having changed, you pass through.
— *I Ching*

In this chapter, we illustrate the application of CDA transformations on some example nested loops. In particular, we describe how the algorithms described in Chapters 5 and 6 are applied to derive suitable CDA transformations, which reduce the number of cache conflicts and the number of ownership tests, respectively. We use the CDA transformed loops to demonstrate that local transformations such as CDA can be useful in reducing the number of cache conflicts and removing ownership tests when it is undesirable to apply global transformations such as array padding and data alignment.

For the purpose of illustration in this chapter, we chose five loops from the Riceps, Arco and SPEC benchmarks [32, 41, 50], and, in particular, loops that come from applications *rtmg*, *mg*, *vpenta*, *swm*, and *wanal*. These loops are appropriate to use as examples, because linear loop transformations alone cannot improve their performance and they thus demonstrate the usefulness of CDA. The loops from *rtmg* and *mg* have a single statement in their loop body, so subexpression-based transformation is essential. The loop in *vpenta* can be transformed both at statement and subexpression granularity. The loops in *swm* and *wanal* can be transformed at statement granularity.

In order to illustrate how the effectiveness of CDA transformations (at loop level) relates to the performance of the entire applications, we also transformed all computation intensive loops in *mg* and *vpenta*.

We compared the execution times of the original and the CDA transformed versions of the loops by running experiments on SUN SPARC 10 and RS/6000 workstations, as well as on the KSR1 multiprocessor. This allows us to show the effect of different cache geometries on the relative performance of the original and the CDA transformed versions of the loops. The SPARC 10

workstation has a 128KB direct-mapped cache; each node of the KSR1 multiprocessor has a 256KB 2-way set associative cache; and the RS/6000 workstation has a 32KB 4-way set associative cache. In addition, we simulated the original and the CDA transformed loops using the XCache simulator from IBM that simulates a RS/6000 with varying cache sizes and geometries [43].[1]

Most of the results we show are for single, individual loop nests. However, it should be noted that the benefits of local transformations such as CDA may be higher than corresponding global transformations when applied to sequences of nested loops, because, global transformations must consider simultaneously a much larger number of constraints arising from all the loop nests. Moreover, these constraints may sometimes prevent the application of global transformations. Overall, we believe it is generally necessary to integrate CDA transformations with other local and global transformation techniques for overall better performance.[2] We believe the key idea should be to use local transformations so that the total number of constraints that a later global transformation must consider is reduced.

8.1 Reducing the Number of Cache Conflicts

We use *rtmg*, *mg* and *vpenta* loops to illustrate the potential performance benefits and potential pitfalls of applying CDA to reduce the number of cache conflicts, relative to the application of array padding. The amount of padding necessary for these loops was determined using the IBM XL compiler. It should be noted that CDA is not capable of removing conflicts that array padding can not. Hence, CDA transformed code can at best perform as well as array padded code. In practice, CDA transformed code performs worse than array padded code because of the extra overheads that CDA introduces. Thus, the application of CDA for reducing the number of cache conflicts is primarily of interest in those situations, described in Section 5.4, where it is undesirable to modify global array layout using array padding.[3] Nonetheless, using *rtmg* and *mg* loops, we show that using CDA can be effective in reducing the number of cache conflicts. Unfortunately, however, the application of CDA can also result in poor performance, due to the introduction of many new references. This is shown using the *vpenta* loop.

[1] The largest direct-mapped cache that can be simulated on the XCache simulator is 64KB. Therefore, the data sizes for the loops were selected to be relatively small so as to obtain simulation results for a range of cache geometries.

[2] For instance, global optimizations to reduce space requirements and cache footprint of the temporary variables introduced help improve the efficacy of CDA transformations.

[3] Some example situations are *i*) when the arrays are accessed in pre-compiled libraries and modules, *ii*) when the arrays are accessed with different shapes in different parts of the program, and *iii*) when programmer specifies that it is unsafe to apply padding.

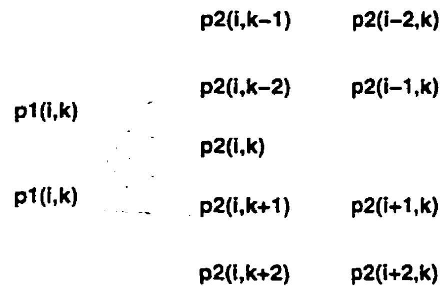


Figure 8.1: Conflicting references in the original rtmg loop.

8.1.1 *Rtmg* Loop

The *rtmg* loop from the Arco Seismic benchmarks suite is a two-dimensional loop with a single statement in the loop body [41]:

```
for i = 3, n-2
   for k = 3, n-2
      p1(i,k) = p2(i,k) - p1(i,k) + p2(i+1,k) + p2(i-1,k) + p2(i,k+1) +
                p2(i,k-1)- p2(i+2,k) + p2(i-2,k) + p2(i,k+2) + p2(i,k-2);
   end for
end for
```

Nested loops such as this are used to implement finite difference operators while migrating seismic sections [41]. References $p1(i,*)$ and $p2(i,*)$ conflict in a direct-mapped cache for certain array sizes, as shown in Figure 8.1. Figure 8.2 shows the number of cache misses when the loop with array sizes of 64x64 is run on a machine with a direct-mapped or 2-way set associative cache of size 16KB and 32KB. Note that that the loop has substantially fewer cache misses in 2-way set associative caches.

We describe here how algorithms A1 and A2 of Chapter 5 can be used to derive a CDA transformation which reduces the number of cache conflicts. Algorithm A1 first decomposes the single statement in the loop body into two new statements so that conflicts occur between statements. Step 3 of Algorithm A1 on page 70 chooses one of the four references, namely $p2(i+1,k)$, $p2(i-1,k)$, $p2(i+2,k)$, and $p2(i-2,k)$, as the first first member of set of partition V_1. After 8 iterations of Step 4, no further vertices can be added to V_1. At this stage, V_1 contains all the $p2$ references. This will create a new statement with a temporary t as the lhs and all references in V_1 on the rhs. The second statement thus contains the $p1$ reference and a reference to temporary t. With this decomposed loop as input, Step 6 of algorithm A2 on page 73 first applies an offset of -1 changing the $p2(i,*)$ references to $p2(i+1,*)$. However, this also changes the $p2(i-1,*)$ references to

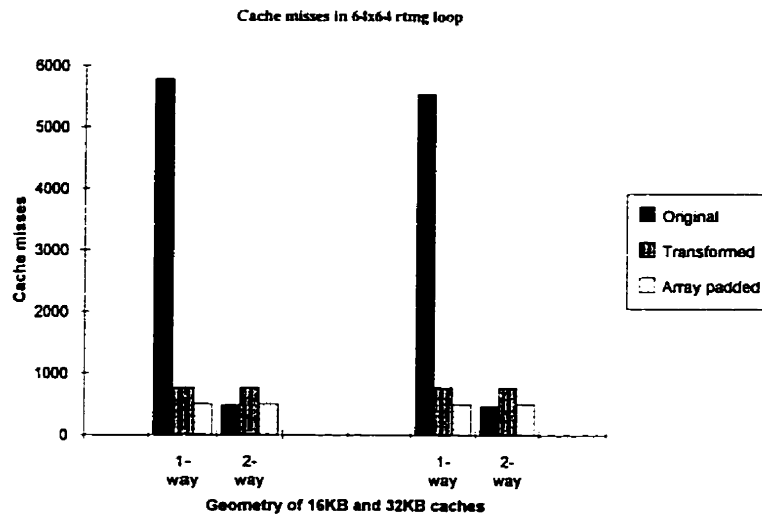


Figure 8.2: The number of cache misses in the original and the CDA transformed *rtmg* loops with array sizes of 64x64.

$p2(i,*)$ which conflict with the $p1(i,*)$ references. Hence, algorithm A2 tries higher offsets and all the conflicts are eliminated when the offset is -3. Thus, the required offset alignment is obtained in 3 iterations of Step 6 of A2. The innermost iterator k was not included in the list of iterators in order to improve the data reuse along cache lines. The size of the temporary array is chosen so that its references do not conflict with $p1$ and $p2$.

The structure of the transformed loop becomes:

```
for i = -1, 2
   for k = 3, n-2
      t(i+3,k) = p2(i+3,k) + p2(i+3,k+1) + p2(i+3,k-1) + p2(i+3,k+2) +
                 p2(i+3,k-2)+ p2(i+4,k) + p2(i+2,k) - p2(i+5,k) + p2(i+1,k);
   end for
end for

for i = 3, n-5
   for k = 3, n-2
      t(i+3,k) = p2(i+3,k) + p2(i+3,k+1) + p2(i+3,k-1) + p2(i+3,k+2) +
                 p2(i+3,k-2)+ p2(i+4,k) + p2(i+2,k) - p2(i+5,k) + p2(i+1,k);
      p1(i,k)  = t(i,k) - p1(i,k) ;
   end for
end for

for i = n-4, n-2
   for k = 3, n-2
      p1(i,k) = t(i,k) - p1(i,k) ;
   end for
end for
```

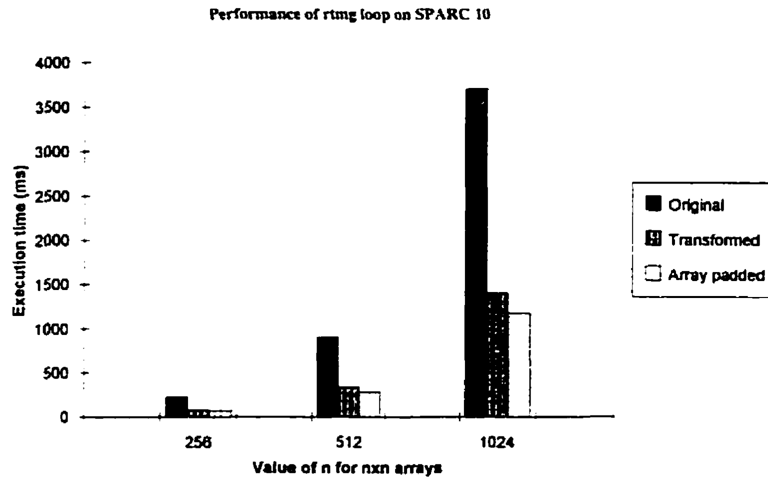


Figure 8.3: Execution time of *rtmg* loop on a SPARC 10 workstation.

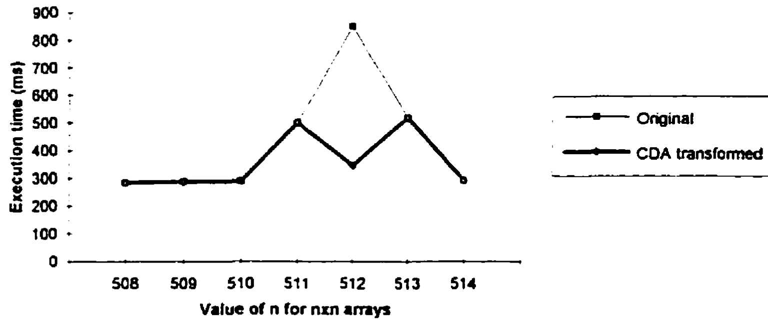


Figure 8.4: Execution time of *rtmg* loop on a SPARC 10 workstation for varying data sizes.

The inner iterations of the transformed loop are conflict free as a result of the CDA transformation. Figure 8.2 shows the reduction in the number of cache misses for 16KB and 32KB caches. Figure 8.3 shows that CDA transformation reduces the execution time to about 35% of the original loop on a SPARC 10 system with 128KB direct-mapped cache.

An equivalent array padded version of the loop has an execution time that is about 30% of the original execution time. This is better than that of the CDA transformed loop, because the CDA transformed loop has overhead by adding references to the temporary array. The CDA transformed loop required about 50% more memory than the original loop due to the introduction of the temporary array.

One of the problems with array padding is that it requires prior knowledge of the array sizes. Often, array sizes are not known in advance, as for example is the case with library routines. In this

case, it is not possible to benefit from any padding. Unaltered library routines can perform poorly, if the parameter data results in excessive cache conflicts. This is illustrated in Figure 8.4 that shows the execution time of *rtmg* loop (in its original form) for varying array sizes. There is a peak where the arrays are 512x512, due to cache conflicts. In contrast to array padding, it is possible to reduce the number of cache conflicts even if the target array sizes are not known at compile time, by using the following strategy. A version of CDA transformed code can be produced, with the assumption of a given array size, and include it *together* with a version containing the original code. Which version to run can then be selected at run-time based on the size of the array. In the example shown in Figure 8.4 a version of CDA transformed code for 512x512 arrays is included and selected if the input parameter has these sizes and the original code is selected in all other cases. The thick line shows the execution time of rtmg loop using this strategy and it is apparent that the peak execution time could be eliminated. Note that the execution time at the remaining two peaks at 511x511 and 513x513 array sizes can be reduced by other CDA transformed versions of the loop.

8.1.2 *Mg* Loop

Mg is an application in the NAS benchmarks suite [11] (and now also included in SPECfp95 [50]), which was designed to demonstrate capabilities of simple multigrid solvers.[4]. Three-dimensional loops such as the one below[5] form the core of the computations. This loop applies an approximate inverse as a smoother [11].

```
for i = 1, I-2
   for j = 1, J-2
      for k = 1, N-2
         U(i,j,k) = U(i,j,k) +
           c(0)*( R(i,j,k) ) +
           c(1)*( R(i-1,j,k) + R(i+1,j,k) + R(i,j-1,k) +
                  R(i,j+1,k) + R(i,j,k-1) + R(i,j,k+1) ) +
           c(2)*( R(i-1,j-1,k) + R(i+1,j-1,k) + R(i-1,j+1,k) + R(i+1,j+1,k) +
                  R(i,j-1,k-1) + R(i,j+1,k-1) + R(i,j-1,k+1) + R(i,j+1,k+1) +
                  R(i-1,j,k-1) + R(i-1,j,k+1) + R(i+1,j,k-1) + R(i+1,j,k+1) ) +
           c(3)*( R(i-1,j-1,k-1) + R(i+1,j-1,k-1) + R(i-1,j+1,k-1) + R(i+1,j+1,k-1) +
                  R(i-1,j-1,k+1) + R(i+1,j-1,k+1) + R(i-1,j+1,k+1) + R(i+1,j+1,k+1) );
      end for
   end for
end for
```

[4]This is a simplified multigrid solver in two important respects: *i*) it solves only a constant coefficient equation, and that only on a uniform cubical grid, *ii*) it solves only a single equation, representing a scalar field rather than a vector field [11]

[5]This loop forms the subroutine called *psinv*.

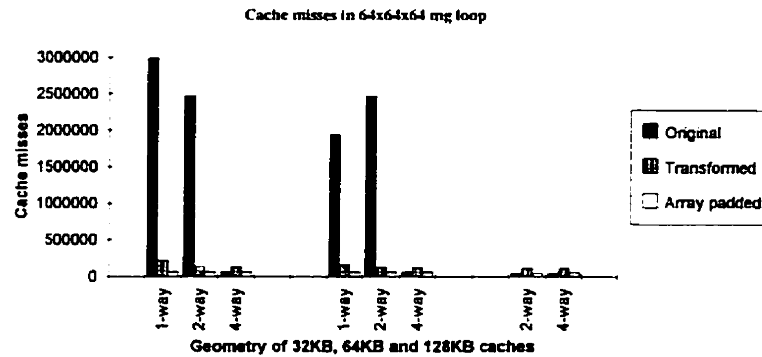


Figure 8.5: Cache misses in the original and the CDA transformed *mg* loop.

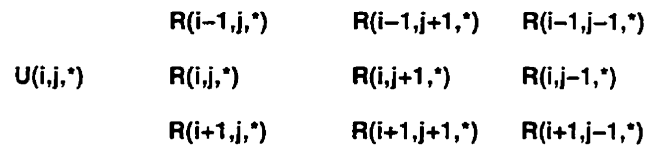


Figure 8.6: Conflict graph for the original *mg* loop.

In a given iteration of the i loop, four "planes" of data, namely $R(i-1,*,*)$, $R(i,*,*)$, $R(i+1,*,*)$, and $U(i,*,*)$, contend for the cache. Figure 8.6 shows the conflicting references when the cache size is equal to the number of elements in a single plane of data.

A CDA that reduces the number of these conflicts is derived as follows. Algorithm A1 decomposes the (only) statement into four statements such that none of the references in the same statement conflict with each other. The four statements correspond to the four independent sets in the conflict graph of Figure 8.6. Derivation of the first three independent sets requires 8 iterations of Step 4 of algorithm A1 on page 70 for each set. The fourth set is formed with the remaining two references. The first statement S_1 has references to the ith plane of R (i.e. $R(i,*,*)$); a second statement S_2 has references to the $(i-1)^{th}$ plane of R (i.e. $R((i-1,*,*)$); a third statement S_3 has references to the $(i+1)^{th}$ plane of R; and the fourth statement S_4 has the reference $U(i,j,k)$ and references to three temporary variables introduced to store the results of the other three statements.

Algorithm A2 then aligns the statements as follows. The iterators are ordered from innermost (k) to outermost (i) iterators — this allows the dimension of the temporary variables to later be reduced from three to two — but the innermost iterator is removed from the candidate iterators in

U(i,j,*)			
	R(i–1,j+5,*)	R(i–1,j+6,*)	R(i–1,j+7,*)
t1(0,j+3,k)			
	R(i,j+2,*)	R(i,j+3,*)	R(i,j+4,*)
t1(0,j+6,k)			
	R(i+1,j+8,*)	R(i+1,j+9,*)	R(i+1,j+10,*)
t1(0,j+9,k)			

Figure 8.7: Conflict graph for the CDA transformed *mg* loop.

order to improve the data reuse along cache lines. Thus, increasing offsets along the j dimension are attempted first for statement S_1. An offset of -1 does not suffice, since references $R(i,j,*)$ would become references $R(i,j+1,*)$ which conflict with references $R(i\pm 1,j+1,*)$ in statements S_2 and S_3. Similarly an offset of -2 does not suffice since references $R(i,j-1,*)$ would become references $R(i,j+1,*)$ which conflict with references $R(i\pm 1,j+1,*)$ in statements S_2 and S_3. The search stops when we arrive at an offset of -3, which eliminates all conflicts due to statement S_1. The derivation of this offset alignment requires 3 iterations of Step 6 of algorithm A2 of page 73. Statements S_2 and S_3 are aligned in a similar way so that they have offsets of -6 and -9, respectively, along the j dimension. Clearly, the derivation of these offset alignments requires 6 and 9 iterations of Step 6 of algorithm A2. Statement S_4 is aligned with the identity transformation. Figure 8.7 shows the conflict graph for the transformed loop; there are no more conflicts in innermost iterations.

The transformed loop is shown below. This loop is guarded; although a guard-free version of the code was used for the experiments, it is not shown here. Since all alignments were along the j dimension, having only three planes of array elements as temporaries, namely $t1(0,*,*)$, $t2(0,*,*)$ and $t3(0,*,*)$ is sufficient. The space overhead for the temporaries is only about 0.5% of the memory required for the original loop.

```
for i = 1, I-2
   for j = -8, J-2
      for k = 1, N-2
         if (j > -3 && j < J-4)
            /* Statement S1 */
            t1(0,j+3,k) =
                        c(0)*( R(i,j+3,k) ) +
                        c(1)*( R(i,j+2,k) + R(i,j+4,k) +
                               R(i,j+3,k-1) + R(i,j+3,k+1) ) +
                    c(2)*( R(i,j+2,k-1) + R(i,j+4,k-1) +
                               R(i,j+2,k+1) + R(i,j+4,k+1) );
        if (j > -6 && j < J-7)
            /* Statement S2 */
            t2(0,j+6,k) =
                        c(1)*( R(i-1,j+6,k) ) +
                        c(2)*( R(i-1,j+5,k) + R(i-1,j+7,k) +
                               R(i-1,j+6,k-1) + R(i-1,j+6,k+1) ) +
                        c(3)*( R(i-1,j+5,k-1) + R(i-1,j+7,k-1) +
                               R(i-1,j+5,k+1) + R(i-1,j+7,k+1) );
        if ( j < J-10)
            /* Statement S3 */
            t3(0,j+9,k) =
                        c(1)*( R(i+1,j+9,k) ) +
                        c(2)*( R(i+1,j+8,k) + R(i+1,j+10,k) +
                               R(i+1,j+9,k-1) + R(i+1,j+9,k+1) ) +
                        c(3)*( R(i+1,j+8,k-1) + R(i+1,j+10,k-1) +
                               R(i+1,j+8,k+1) + R(i+1,j+10,k+1) );
        if (j > 0 && j < J-1)
            /* Statement S4 */
            U(i,j,k) = U(i,j,k) + t1(0,j,k)+ t2(0,j,k)+ t3(0,j,k);
      end for
   end for
end for
```

Figure 8.5 shows the number of cache misses obtained in a simulation of the loop with arrays U and R having size 64x64x64 for cache sizes 32KB, 64KB and 128KB, with a 128 byte cache line and associativity varying from 1 to 4. The figure shows that the CDA transformed loop has substantially fewer cache misses in most cases. However, for large caches (128KB in this case) and 4-way set associative caches, the CDA transformed loop has more misses than the original loop. The CDA transformed loop has 20% more references in total than the original loop.[6] Moreover, the original loop does not have conflicts in 4-way set associative caches, since the loop has only four planes of arrays conflicting for the cache. When the cache size is large, say 128KB, then all four planes of the arrays fit in the cache in this example.

Figure 8.8 compares the number of additional array elements required for a CDA transformed *mg* loop with that required for an array padded *mg* loop. In general, the CDA transformed loop

[6] Figure 8.5 does not show the total number of references.

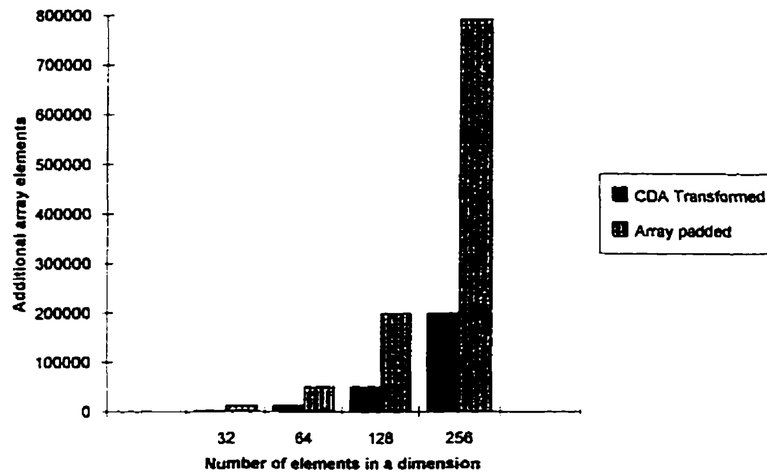


Figure 8.8: Number of additional array elements in the CDA transformed and the array padded *mg* loops.

requires substantially fewer additional array elements when the array dimensions are high.

Figure 8.9 shows the improvement in execution time of the *mg* loop with 256x256x256 arrays on a SPARC 10 workstation and a single KSR1 processor. The CDA transformed loop ran faster than the original loop by a factor of 5-6 on both platforms, although slower than but within 20%-25% of the array padded version of the loop. The difference between CDA and array padded code is that the CDA code still has cache conflicts in the iterations of the outer iterator i and increased number of references due to the temporary variables that were introduced. For example, conflicts between iterations in the outer loop exist because the array elements in the $R(i+1,*,*)$ plane accessed in iterations $(i,*,*)$ are no longer in the cache when accessed in iterations $(i+1,*,*)$, having been displaced by $R(i,*,*)$ in $(i,*,*)$ iterations.

We transformed the three dimensional loops in both *psinv* and *resid* subroutines, which account for 90% of the computation in the benchmark. The transformed application ran 2.5 times faster than the original version on a SPARC 10 workstation when the subroutines had cubical grid sizes that create cache conflicts.[7] It is interesting to note that a padding algorithm will choose not to pad the arrays considering the entire application, since the subroutines access the arrays in a shape different from the declared shape in common blocks.

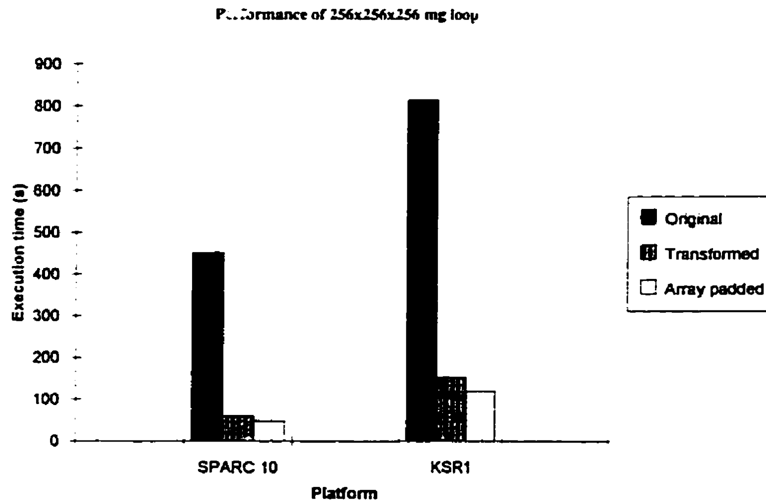


Figure 8.9: Execution time of *mg* loop with 256x256x256 arrays on a SPARC 10 workstation and a single KSR1 processor.

8.1.3 *Vpenta* Loop

Vpenta is part of NASA7 in the SPECfp92 benchmark suite [50]. The programs in NASA7 were designed to represent typical numerical applications in engineering. Vpenta is designed to simultaneously invert 3-dimensional pentadiagonals [50]. Vpenta consists of two similar 2-dimensional loops, one of which is shown below:[8]

```
for j = 0, n
   for k = 2, n-2
      RLD2 = .01*A(k,j);
      RLD1 = .01*(B(k,j) - RLD2*X(k,j-2));
      RLD = C(k,j) - (RLD2*Y(k,j-2) + RLD1*X(k,j-1));
      RLDI = .000000001*1.0/RLD;
      F(0,k,j) = (F(0,k,j) - RLD2*F(0,k,j-2) - RLD1*F(0,k,j-1))*RLDI;
      F(1,k,j) = (F(1,k,j) - RLD2*F(1,k,j-2) - RLD1*F(1,k,j-1))*RLDI;
      F(2,k,j) = (F(2,k,j) - RLD2*F(2,k,j-2) - RLD1*F(2,k,j-1))*RLDI;
      X(k,j) = (D(k,j) - RLD1*Y(k,j-1))*RLDI;
      Y(k,j) = E(k,j)*RLDI;
   end for
end for
```

Cache locality can be improved in this loop by performing a loop interchange (which can be done with a simple linear loop transformation), as all elements of the cache lines would then be accessed in consecutive iterations. However, for certain cache geometries, the cache performance can continue to be unsatisfactory due to cache conflicts. Figure 8.10 shows the number of cache

[7] Due to relatively high memory requirement of the application, we ran it only with 128x128x128 sized arrays.
[8] The 3-dimensional array F is of size 3x128x128 and the other arrays are of size 128x128.

misses that occur when the loop is executed with arrays of size 64x64 on a machine with 32KB, 64KB and 128KB caches and associativity varying from 1 to 4. From the figure, it is clear that increasing associativity and the size of the cache have only a moderate effect on the number of cache misses if the loop is not transformed. Here, we show how CDA can be applied to reduce the number of conflicts in the interchanged loop.

This is an example where array padded version performs much better than the CDA transformed version, since CDA transformation introduces a large number of temporary arrays. The space overhead of the transformed loop is nearly 60% of the memory required for the original loop. For the experiment we did not decompose the statements, because algorithm $A1$ would be over optimistic and decompose the first, second, third, eighth, and the ninth statement, causing each new statement to have a single array reference on the right hand side.

The first four statements have scalars on the lhs; these are expanded into 2-dimensional arrays so as to enable alignment. As before, dimension j was not included in the list of iterators to consider for alignment so as maximize cache line reuse. Therefore, all alignments are along dimension k. Also, we consider, the statements for alignment in the order they appear in text.

The reference to A in the first statement does not conflict with other references when the statement is applied an offset of -1. It is not legal to apply either a positive or a negative offset to the second, third and the fourth statements, due to dependences on variables $RLD2$, $RLD1$ and RLD. It is also not legal to apply a negative offset alignment to the fifth statement (due to dependences on $RLD2$, $RLD1$ and $RLDI$). A positive offset of 1 removes conflicts due to $F(0, k, *)$, making them $F(0, k-1, *)$. Statements six through nine are similar to the statement five in that they cannot be applied negative offset alignment. Hence, they are applied increasing offsets of 2 through 5. The resulting CDA transformed loop is shown below. A guard-free version of this case is used in the experiments.

```
for k = 1, n+3
   for j = 0, n
     if(k < n-2)
       RLD2(k+1,j) = .01*A(k+1,j);
     if (k > 1 && k < n-1)
       RLD1(k,j)   = .01*(B(k,j) - RLD2(k,j)*X(k,j-2));
     if (k > 1 && k < n-1)
       RLD(k,j)    =  C(k,j) - (RLD2(k,j)*Y(k,j-2) + RLD1(k,j)*X(k,j-1));
     if (k > 1 && k < n-1)
       RLDI(k,j)   = .000000001*1.0/RLD(k,j);
     if (k > 2 && k < n)
       F(0,k-1,j)  =  (F(0,k-1,j) - RLD2(k-1,j)*F(0,k-1,j-2) -
                       RLD1(k-1,j)*F(0,k-1,j-1))*RLDI(k-1,j);
     if (k > 3 && k < n+1)
       F(1,k-2,j)  =  (F(1,k-2,j) - RLD2(k-2,j)*F(1,k-2,j-2) -
                       RLD1(k-2,j)*F(1,k-2,j-1))*RLDI(k-2,j);
     if (k > 4 && k < n+2)
       F(2,k-3,j)  =  (F(2,k-3,j) - RLD2(k-3,j)*F(2,k-3,j-2) -
                       RLD1(k-3,j)*F(2,k-3,j-1))*RLDI(k-3,j);
     if (k > 5 && k < n+3)
       X(k-4,j)    =  (D(k-4,j) - RLD1(k-4,j)*Y(k-4,j-1))*RLDI(k-4,j);
     if (k > 6 && k < n+4)
       Y(k-5,j)    =   E(k-5,j)*RLDI(k-5,j);

   end for
end for
```

The number of cache misses in the CDA transformed loop is shown in Figure 8.10. For all cache geometries, the transformed loop has substantially fewer cache misses than the original loop. The execution times of the original, the CDA transformed, and an array padded (also loop interchanged) code are shown in Figure 8.11 for the SPARC 10, KSR1 and RS/6000 platforms. The improvement in execution times obtained by CDA transformation is not as high as might be expected when considering the reduced number of cache misses. The reason for this is that scalar expansion nearly doubles the number of total references. Moreover, the CDA transformed code requires substantially more memory than the corresponding array-padded code: Figure 8.12 compares the number of array elements that are added for the CDA code to the number of elements added to the array padded code. The CDA transformed loop would require substantially fewer additional array elements if the loop were aligned along the j iterator (instead of the k iterator), but the execution time would then be even higher because of the poor reuse of the cache lines.

Finally, we transformed the two-dimensional loops that together account for over 90% of the execution time of *vpenta*. The transformed version of the entire *vpenta* program ran 1.5 times faster than the original program. In comparison, the array padded version ran about 2.7 times faster than

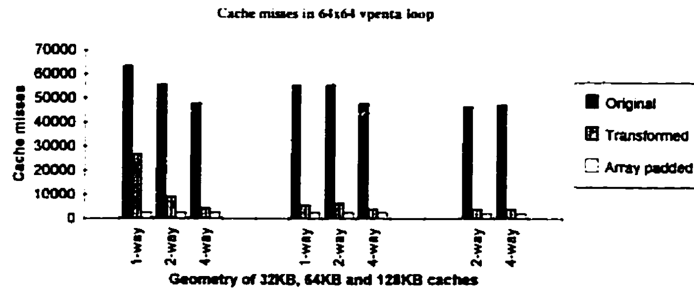


Figure 8.10: Cache misses in the original and the CDA transformed *vpenta* loop.

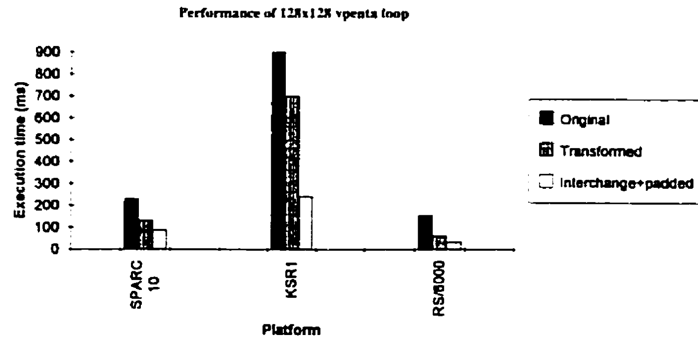


Figure 8.11: Execution time of *vpenta* loop on SPARC 10 workstation, a processor of the KSR1 and RS/6000 workstation.

the original loop.

8.2 Removing Ownership Tests

We use *wanal* and *swm* loops to illustrate the potential performance benefits of applying CDA to remove the ownership tests. The performance improvements are comparable to the performance improvements due to guard elimination.[9]

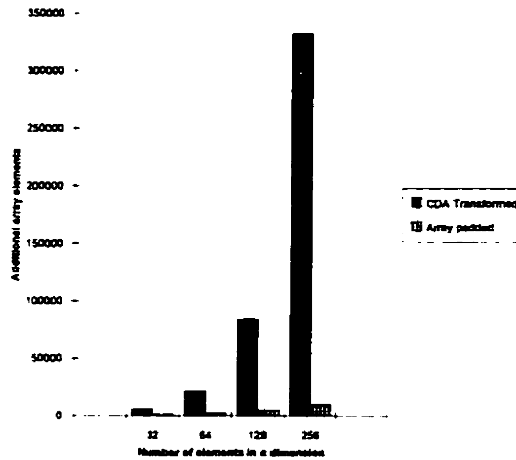


Figure 8.12: Number of additional array elements required for the CDA transformed and the array padded *vpenta* loops.

8.2.1 *Wanal* Loop

Wanal is a wave equation solver that is part of the Riceps benchmark suite [32]. The three-dimensional loop from the benchmark shown below has two statements in its loop body.

```
l = max(p*b, 0); u = min((p+1)*b-1, M+1);
for i = M+1, 0, -1
   for j = M+1, 0, -1
      for k = 0,1
         S1: if (i > l-1 && i < u)
                EL(2*i,2*j,k)=(EL(2*i+1,2*j,k)+EL(2*i-1,2*j,k))/2
         S2: if (j > l-1 && j < u)
                EL(2*j,2*i-1,k)=(EL(2*j,2*i,k)+EL(2*j,2*i-2,k))/2
      end for
   end for
end for
```

This code performs poorly when blocks of array planes $EL(i,*,*)$ are mapped onto the processors and the statements are mapped onto the processors using the owner-computes rule: The array is accessed in such a way that each processor owns the lhs of both statements in only a small number of iterations. Thus, each processor must execute every iteration of the loop to determine whether it owns one of the lhs array elements. It is not possible to apply data alignment in this case, because the lhs references are to the same array.

In applying CDA, it is not necessary to decompose the statements since the statements are mapped in their entirety. Algorithm *B2* identifies two alignment transformations for each statement: first the identity transformation and a transformation that interchanges the i and j iterators

[3]The application of CDA can potentially improve the later communication optimizations. These improvements result from increased opportunities for vectorizing communications.

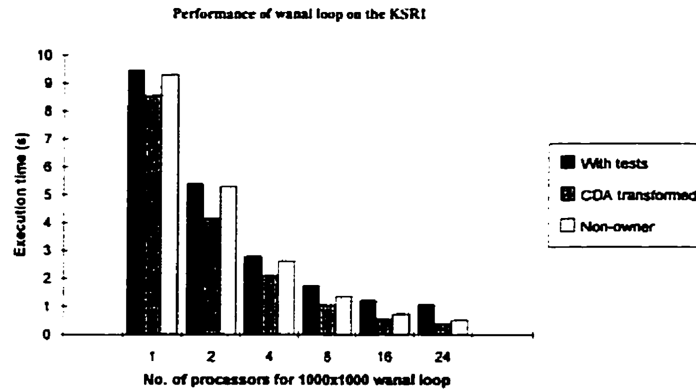


Figure 8.13: Execution time of *wanal* loop on KSR1.

to align the lhs of the statements. Both alignment transformations are legal.

The CDA transformed loop where statement S_1 is applied the identity transformation and statement S_2 is applied an interchange of the i and j iterators is shown below:

```
l = max(p*b, 0); u = min((p+1)*b-1, M+1);
for i = u, l, -1
   for j = M+1, 0, -1
      for k = 0,1
         S1: EL(2*i,2*j,k)=(EL(2*i+1,2*j,k)+EL(2*i-1,2*j,k))/2
         S2: EL(2*i,2*j-1,k)=(EL(2*i,2*j,k)+EL(2*i,2*j-2,k))/2
       end for
   end for
end for
```

In this SPMD code, there no need for ownership tests, and each processor executes just a smaller subset of the entire iteration space. Figure 8.13 compares the execution time of the CDA transformed loop with the execution times of the *wanal* loop with ownership tests. The CDA transformed loop improved the execution time by between 10% and 60% over with the number of processor varying from 1 to 24. Since the KSR1 has a shared address space, the processor owning the lhs of statement S_1 in an iteration can be forced to execute statement S_2 as well, even though it is not the owner. For completeness, we show the execution time of this case by the bars labeled *Non-owner* in Figure 8.13. For this experiment, the bounds of the i iterator were changed so as to scan the local iteration space for S_1. Hence, this version of the loop does not use any tests in the loop body. The CDA transformed loop improved the execution time by about 10-25% over the this version of the loop. This loop performs worse than the CDA transformed loop since it has poorer cache locality while accessing array elements for statement S_2.

8.2.2 *Swm* Loop

Swm is the shallow water program of the SPEC benchmark suite designed for weather prediction [50]. It consists of subroutines *calc1* and *calc2* which contain similar nested loops that need data alignments in order to eliminate ownership tests if we assume owner-computes rule. Here, we use the loop in the *calc*1 subroutine to show how CDA transformation can also remove ownership tests.

This loop, here called, the *swm* loop, has 4 statements, with lhs references $cu(i+1,j)$, $cv(i,j+1)$, $z(i+1,j+1)$ and $h(i,j)$. Assume the SPMD code with the arrays mapped onto processors by blocks of rows of size b. In the *swm* loop shown below, expressions l and u for processor p represent the rows of the lhs arrays mapped onto the processor.

```
l = max(p*b, 1); u = min((p+1)*b-1, n-1);
for i = l-1, u
   for j = 1, n-1
      if (i < u)    cu(i+1,j)   = 0.5*(p(i+1,j)+p(i,j))* u(i+1,j);
      if (i > l-1)  cv(i,j+1)   = 0.5*(p(i,j+1)+p(i,j))* v(i,j+1);
      if (i < u)    z(i+1,j+1) = fsdx*(v(i+1,j+1) - v(i,j+1)) - fsdy*(u(i+1,j+1) -
                                 u(i+1,j))/p(i,j) + p(i+1,j)+p(i+1,j+1)+p(i,j+1);
      if (i > l-1)  h(i,j)      = p(i,j) + 0.25*(u(i+1,j)*u(i+1,j)+ u(i,j)*u(i,j) +
                                 v(i,j+1)*v(i,j+1) + v(i,j)+v(i,j));
   end for
end for
```

The ownership tests for each statement appear as conditionals using the value of the i iterator. Due to the $(i+1,*)$ references to cu and z on the lhs of the first and third statement, processors must execute iterations i, where $l-1 \leq i \leq u$; Statements 1 and 3 get executed in iterations $l-1 \leq i < u$ and statements 2 and 4 get executed in iterations $l-1 < i \leq u$.

This SPMD code incurs two overheads: first the execution of tests in every iteration, and second the execution of one i iteration more than necessary. The ownership tests can be eliminated by data alignments which co-locate $cu(i+1,*)$, $cv(i,*)$, $z(i+1,*)$, and $h(i,*)$.

In applying CDA transformation as an alternative to data alignment, it is not necessary to apply algorithm *B1* to decompose the statements, since each statement of the loop is mapped in its entirety. Algorithm *B2* identifies four candidate alignment transformations for each statement: the identity transformation and three transformations each aligning the statement to one of the other three statements.[10] These alignments attempt to align the statements such that all the lhs

[10] There are only two unique alignment transformations for each statement.

references are either of the form $(i, *)$ or of the form $(i+1, *)$. In this case, all candidate alignments are legal.

The CDA transformed code where the statements are aligned such that the lhs references are $cu(i, j)$, $cv(i, j+1)$, $z(i, j+1)$, and $h(i, j)$, and where the guards have been removed is shown below:

```
if (p == 1)
   for j = 1, n-1
      cv(1,j+1) = 0.5*(p(1,j+1)+p(1,j))* v(1,j+1);
      h(1,j)    = p(1,j) + 0.25*(u(2,j)*u(2,j)+ u(1,j)*u(1,j) +
                  v(1,j+1)*v(1,j+1) + v(1,j)+v(1,j));
   end for
end if

// begin core

l = max(p*b, 2); u = min((p+1)*b-1, n-1);
for i = l, u
   for j = 1, n-1
      cu(i,j)   = 0.5*(p(i,j)+p(i-1,j))* u(i,j);
      cv(i,j+1) = 0.5*(p(i,j+1)+p(i,j))* v(i,j+1);
      z(i,j+1)  = fsdx*(v(i,j+1) - v(i-1,j+1)) - fsdy*(u(i,j+1) - u(i,j))/
                  p(i-1,j) + p(i,j)+p(i,j+1)+p(i-1,j+1);
      h(i,j)    = p(i,j) + 0.25*(u(i+1,j)*u(i+1,j)+ u(i,j)*u(i,j) +
                  v(i,j+1)*v(i,j+1) + v(i,j)+v(i,j));
   end for
end for

// end core

if ( p == P)
   for j = 1, n-1
      cu(n,j)   = 0.5*(p(n,j)+p(n-1,j))* u(n,j);
      z(n,j+1)  = fsdx*(v(n,j+1) - v(n-1,j+1)) - fsdy*(u(n,j+1) - u(n,j))/
                  p(n-1,j) + p(n,j)+p(n,j+1)+p(n-1,j+1);
   end for
end if
```

In this SPMD code, there is no need for ownership tests inside the loop bodies, since each statement of the loop now accesses the rows of the arrays mapped onto the same processor. The bounds of the i iterator in the core of the transformed loop were modified to be l and u so that the loop scans the local iteration space of the processors. Note that the loop nests outside the core of the transformed loop will still need one test each.

Figure 8.14 compares the execution time of the CDA transformed loop with the execution times of the *swm* loop with ownership tests and with the execution of the array aligned *swm* loop. The

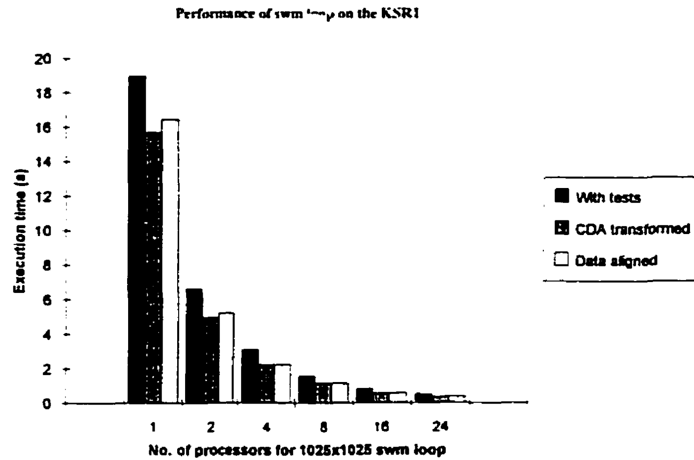


Figure 8.14: Execution time of *swm* loop on KSR1.

three versions of the loop were executed on the KSR1 multiprocessor using the NUMACROS macro package to execute the i loop in parallel [33]. The CDA transformed loop improves the execution time by 25-30% over the case with ownership tests for 1 to 24 processors. The CDA transformed loop performed slightly (1-5%) better than the data aligned loop. This improvement is mainly due to distributing the computations of the statements into three subnests.

Chapter 9

Concluding Remarks

Woods are lovely, dark and deep
I have promises to keep
And miles to go before I sleep
And miles to go before I sleep
— *Robert Frost*

9.1 Summary

The main focus of this dissertation has been the design of the CDA transformation framework that extends the linear loop transformation framework by a significant step. While, the linear loop transformation framework has already been very effective in exposing parallelism and improving memory and cache locality, our goal was to extend and enhance the capabilities of the linear loop transformation framework. In particular, this dissertation makes the following contributions.

Granularity of loop transformations

First, we have shown that transformations of nested loops at statement and subexpression granularity has potential with respect to performance. The linear loop transformation framework cannot transform at this granularity, since it can only transform at the granularity of entire iterations. A facility to transform loops at statement and subexpression granularity extends the linear loop transformation framework, because it allows the composition of the iterations to be changed as well as the execution order of the re-composed iterations.

CDA transformation framework

We have described the CDA transformation framework, which has two components: Computation Decomposition and Computation Alignment. Computation Decomposition partitions the iteration space into possibly several computation spaces, each representing the computations of a statement

or a subexpression. Computation Alignment then applies a separate linear transformation to each of the computation spaces. The transformed computation spaces are fused together to define the new iterations. In the CDA framework, linear loop transformations are just a special case, where each computation space is transformed the same way.[1]

Unfortunately, the price of additional flexibility in CDA transformation comes at the cost of additional execution and space overheads in the transformed loops. The overheads arise in the form of empty iterations, guard computations and space for temporary variables. We described techniques that improve the efficiency of CDA transformed loops by reducing these overheads. Simple transformations of the computation spaces, such as small offset alignments, tend to introduce lower overheads than transformations which are defined by general non-singular integer matrices.

Opportunities for CDA transformations

We have identified several types of optimizations that can benefit from CDA. These optimizations are either new transformations or generalizations of existing transformations. For example, the CDA transformations for reducing the number of cache conflicts and the CDA transformations for removing ownership tests are new in that they achieve by code transformations what has traditionally been achieved by data transformations; hence local transformations can now achieve what traditionally was achieved with global transformations. Similarly, the CDA transformations for improving instruction level parallelism are new in that they modify the composition of iterations through relatively high order transformations. As an example of how CDA generalizes existing transformations, the CDA framework unifies loop distribution and loop alignment transformations into a single linear algebraic formalism. The CDA framework generalizes loop distributions in that it can effect partial loop distributions, and it generalizes loop alignments in that it allows the alignment of statements by non-singular integer matrices.

Automatic derivation of transformations

The key to the success of a transformation framework is the availability of techniques to automatically derive transformations appropriate for a given loop. The relatively fine-grained restructuring that is possible within the CDA framework implies vastly larger search spaces for deriving transformations than those that exist when deriving a linear loop transformation. It is, therefore, necessary to use heuristics to derive CDA transformations efficiently, as an exhaustive search would be com-

[1] Hence, Computation Decomposition is a redundant step for a linear transformation.

putationally intractable. These heuristics must make use of the knowledge about the optimization context in order to be effective.

In deriving these heuristics, we attempt, whenever possible, to build on techniques that already exist in other frameworks. For example, a CDA transformation to reduce cache conflicts attempts to move conflicting references in an iteration away from each other and into adjacent iterations. That is, conflicting memory references are moved away from each other in time, which is analogous to the way array padding algorithms move array elements that are the target of conflicting accesses away from each other in space. Similarly, a CDA transformation to remove ownership tests attempts to move computations that are mapped onto the same processor together from different iterations into the same iteration. We derive appropriate transformation matrices for this purpose similar to the way data alignment modifies two array references to be similar.

Experimental demonstration

Lastly, we have illustrated how CDA transformations can be derived for example nested loops using techniques discussed in this dissertation. The experiments demonstrate that local transformations such as CDA can be useful in reducing the number of cache conflicts and removing ownership tests, when it is undesirable to apply global transformations such as array padding and data alignment. The experimental results also demonstrate that it is necessary to apply CDA carefully, since the overheads introduced by CDA, at times, also reduce performance substantially. In our experiments, we have focused on those loops that require statement and subexpression level restructuring, because linear loop transformations alone cannot help improve performance in these cases.

9.2 Future Work

In exploring CDA, we have just scratched the surface. Much work is still necessary to understand the full potential of this framework. The work described in this dissertation can be directly extended in a number of directions.

Techniques for deriving CDA transformations

The techniques we described for deriving CDA transformations demonstrate that simple heuristics may often suffice, even if CDA transformations are complex relative to linear loop transformations. However, these heuristics can be improved further and extended. For example, the algorithm to

derive CDA transformations for reducing cache conflicts can be extended to *i*) reduce conflicts from outer loop iterations (as well as innermost iterations), *ii*) align statements in an order determined by dependences, *iii*) re-consider alignments for statements dependent on a statement S when the candidate alignment for S is illegal, and *iv*) test for conflicts by symbolic pattern matching. The algorithm to derive CDA transformations for removing ownership tests can be extended to consider only those candidate alignments which preserve the rank of the dependence matrix (and hence parallelism).

For the purpose of applying CDA to improve instruction level parallelism, it is necessary to find heuristics to recognize iteration compositions that have improved parallelism. Moreover, we believe that partial distributions are a very useful generalization of loop distribution; further study is required to identify situations where partial distributions improve performance and to be able to automatically derive appropriate CDAs for this purpose.

Duality of transformations

We showed that certain CDA transformations can be viewed as duals of certain data transformations, in particular array padding and data alignment. The data transformations and the corresponding CDAs each have their own advantages and disadvantages. We believe that an integrated approach is better than using an exclusive-or approach. Further investigation is required to find strategies for this integrated approach. It is also possible to CDA transform loops so that iterations access data at cache line and page boundaries when the arrays are not explicitly aligned to these boundaries.

Unified approach to optimizations

The execution time of a nested loop can be improved by applying loop transformations that improve features of the loop such as cache access behavior, parallelism, instruction level parallelism, and load balance. However, a loop transformation may improve one feature of the loop, but may at the same time, worsen other features of the loop. Traditionally, such compatibility issues are resolved by identifying features that most affect the loop performance and applying the loop transformations in a fixed order. For example, when cache conflicts cause serious performance degradation, then a transformation that reduces the number of cache conflicts may still significantly improve execution time, even though the loop may continue to have ownership tests. It is interesting to model nested loops and machine architectures so that the impact of a transformation could be accurately deter-

mined directly in terms of expected execution time of loops. It may seem that such an approach makes it difficult to ensure reasonable compilation times. However, recent trends in analysis and optimization techniques indicate that such an approach may have acceptable average time complexity [45]. The algorithm recently proposed by Lim and Lam to derive the optimal transformation that maximizes parallelism and minimizes the degree of synchronization can be viewed as indicative of techniques that will be designed in the future capable of deriving transformations optimal for given target architectures [38].

CDA as a generalized transformation framework

Clearly, a transformation framework is only as good as the accompanying algorithms that can derive appropriate and effective transformations. Much further work is necessary in improving the algorithms described here and designing new algorithms for the other applications mentioned. Moreover, although we found that simple heuristics that were natural for the optimization in question were often quite effective, it is still necessary to test these heuristics on a wider variety of nested loops to fully understand their usefulness and robustness.

The most appropriate way to do this is to integrate the CDA framework into an existing compiler framework so that the ideas presented in this dissertation can be tested on the huge scientific and engineering code base that already exists. The prospects for doing this have improved over the last year or two with newer analysis and code generation techniques. However, the techniques used in the CDA framework are much more involved than the techniques used in the linear loop transformation framework, and the CDA transformed loops tend to be more complex and larger in size. Hence, it might be useful to identify a subset of the CDA framework that can be integrated with current compiler implementations with less effort.

9.3 Epilogue

It should be noted that it took the compiler community over five years and a huge amount effort to fully realize the potential and develop efficient techniques for the linear loop transformation framework to the point where they could be integrated into today's compilers. CDA will require a much larger effort if its potential is to be fully exploited. We believe that it is important to invest in this effort, because much more aggressive compiler techniques for restructuring programs will become necessary in order to deal with the complexities that will be introduced with future advances

in computing technology and architectures. Techniques such as CDA, as well as other non-linear code and data transformations, will play a prominent role in the set of aggressive transformation techniques that will become necessary to improve performance on future systems.

APPENDIX A

A Catalog of Loop Transformations

We classify the transformations into *preliminary*, *primary*, and *secondary* transformations. Preliminary transformations are the first to apply, and the purpose is to improve the data and control dependence analysis. Primary transformations achieve the intended restructuring of the program. Some of the objectives could be to enhance parallelism, load balance and/or locality. Secondary transformations improve the performance of the programs further by fine tuning.

The loop transformations can change the dependence structure. Therefore, it is necessary to ensure that a transformed loop preserves the sequential semantics, that is, produces the same results as the original loop. A transformation is said to be legal if all the dependences in the transformed loop are positive.

A.1 Preliminary Transformations

These transformations are intended to improve further analysis.[1]

Induction variable elimination

Simplifies the subscript analysis in dependence tests.

```
j = n
for i = 0, n                          for i =0, n
  A(i) = B(j-1)                         A(i) = B(n-i-1)
  B(j) = C(i)       ===>                B(n-i) = C(i)
  j = j-1                             end for
end for
```

[1] Although most of the preliminary transformations were introduced in the context of vectorization, we leave out vectorization techniques here.

Normalization

Many transformations assume that the lower bound of a loop index starts with zero and has a stride of one. [2] Normalization makes a loop nest to conform to this assumption.

```
for i = 2, n, 2                         for i = 0, n/2-1
  for j = 0, n                            for j = 0, n
    A(i,j) = A(i-2,j)    ====>              A(2i+2,j) = A(2i,j)
  end for                                 end for
end for                                 end for
```

Forward substitution

Simplifies dependence analysis by substituting constants for the expressions in array references.

```
x = n +1
for i = 0, n                            for i =0, n
  A(i) = B(i)                             A(i) = B(i)
  C(i) = A(x)          ===>               C(i) = A(n+1)
end for                                 end for
```

False dependence elimination

Anti and output dependences are false as they arise just because of sharing the same storage location. Idea is to eliminate these and work only with the flow (true) dependence.

```
A = B + C                     A1 = B + C
A = D + E          ===>       A2 = D + E
... = A                       ... = A2
```

Loop distribution or Fission

In order to enable vectorization, or provide simpler loops for analysis, it distributes statements in a loop into multiple similar loops.

```
for i = 0, n                            for i = 0, n
```

[2] As we see later, changing the stride itself is treated as another transformation called scaling.

```
    A(i) = x                         A(i) = x
    C(i) = A(i-1)+D(i-1)     ===>   end for
    D(i) = C(i)                     for i = 0, n
end for                               C(i) = A(i-1)+D(i-1)
                                      D(i) = C(i)
                                    end for
```

Node splitting

Dependence cycles in a loop, prevent loop distribution and vectorization. It is possible to break the cycles by introduction of temporary variables to keep copies of data. Node splitting is essentially an introduction of temporary variable followed by loop distribution.[3]

```
for i=1,n                                for i = 1,n
  A(i) = B(i) + C(i)                      A(i) = B(i) + C(i)
  D(i) = A(i-1) + A(i+1)   ===>           Temp(i) = A(i+1)
end for                                   D(i) = A(i-1) + Temp(i)
                                         end for
```

The original loop contains a flow dependence (1) and an anti-dependence (-1) leading to a cycle, and preventing the loop distribution. Introduction of temporary variable Temp to keep a copy of A enables us to distribute the loop (and subsequently vectorize) as below.

```
for i = 0,n
  A(i) = B(i) + C(i)
  Temp(i) = A(i+1)
end for
for i = 0,n
  D(i) = A(i-1) + Temp(i)
end for
```

Loop fusion

The opposite of distribution (or fission). Fusing adjacent loops increases the grain size and decrease the overhead of a do-all loop. This can also improve cache performance by increasing reuse of

[3] Some consider only the introduction of temporaries as node splitting.

elements accessed in fused loops.

A.2 Primary Transformations

One may have to apply a sequence of these transformations.

Strip mining

Strip mining transforms a 1-d loop into a 2-d loop.[4] A dependence (c) becomes $(0, c)$ and $(1, c - S - 1)$, where S is the strip size (and assuming $S \geq c$). It is always legal to do strip mining. (That is, strip mining does not result in negative dependences in the transformed loop.) The bounds are rectangular and easy to compute. Strip mining is done mostly to exploit the vector register size.

```
for i = 0, n                          for ii = 0, n, 64
  A(i) = A(i-1)        ===>             for i = ii, min(ii+63, n)
end for                                   A(i) = A(i-1)
                                        end for
                                      end for
```

Loop interchange

Interchanges nested loops.[5] A dependence (a, b) becomes (b, a), and thus interchange is not valid if any dependence (a, b) has a negative b. Loop interchange can enhance inner loop parallelization, vectorization, outer loop parallelization, adjustment of array access stride, and match loop parallelism to data distributions. Loop bound computation can be non-trivial if the iteration space is non-rectangular.

```
for i = 0, n                          for j = 0, n
  for j = 0, n                          for i = 0, n
    A(i,j) = A(i,j-1)    ====>            A(i,j) = A(i,j-1)
  end for                               end for
end for                               end for

for i = m1, n1                        for j = m2, n1
```

[4] We see later in the document that strip mining is a trivial case of tiling.

[5] In a doubly nested loop this corresponds to transposing the iteration space.

```
  for j = m2, i                         for i = max(m1,j), n1
    A(i,j) = A(i,j-1)    ====>            A(i,j) = A(i,j-1)
  end for                                end for
end for                                end for
```

Loop permutation

Generalization of a loop interchange. Gives a loop that has indices that are a permutation of the original loop indices. A dependence (a, b, c) with permutation $\begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix}$ becomes (c, a, b). A permutation is valid if all the permuted dependences are positive. The above permutation is valid if the dependences are $\{(1, 1, 0), (1, -1, 1)\}$, not if a dependence $(1, 0, -1)$ also exists. Loop bounds are again non-trivial to compute for non-rectangular iteration spaces. A permutation can be realized as a sequence of interchanges, but finding legal sequences of interchanges is a difficult task.

```
for i = 0, n                                      for k = 0, n
  for j = 0, n                                      for i = 0, n
    for k = 0, n                                      for j = 0, n
      A(i,j,k) = A(i-1,j+1, k-1)   ===>                 A(i,j,k) = A(i-1,j+1,k-1)
    end for                                           end for
  end for                                           end for
end for                                           end for
```

Loop Reversal

Reversing a loop may be necessary to enable loop interchanges. By reversal of a loop with lower bound L and upper bound U, it is made to iterate from $-U$ to $-L$ with the same stride. A dependence (a, b) becomes $(a, -b)$. Note that, whenever there are dependences carried by the outermost loop, it is illegal to reverse it. For example, a dependence $(1, 2)$ becomes $(-1, 2)$ after interchanging the outer loop, which is illegal as the new dependence is negative.

```
for i = 0, n                           for i = 0, n
  for j = 0, n                           for j = -n, 0
    A(i,j) = A(i-1,j+1)    ====>            A(i,-j) = A(i-1,-j+1)
```

```
  end for                              end for
end for                              end for
```

Loop skewing

A dependence $(d1, d2)$ becomes $(d1, fd1 + d2)$ where f is the skew factor. A valid skew can always be found for a given loop.

```
With a skew factor of +1.
for i = 0, n                         for i = 0, n
  for j = 0, n                         for j = i, i+n
    A(i,j) = A(i-1,j)    ====>           A(i,j-i) = A(i-1,j-i)
  end for                              end for
end for                              end for
```

Wavefront

The idea is to find a family of hyperplanes in the iteration space along which the iterations can be executed in parallel. The loop is transformed in such a way that the hyperplanes are executed sequentially, and all iterations on a hyperplane are executed in parallel (i.e. inner loop parallelization). A wavefront transformation results in dependences carried by the outer most loop. A valid wavefront can always be found.

Loop skewing is just a simple instance of wavefront transformation.

Unimodular and Linear loop transformations

A unified transformation that subsumes permutation, reversal, and skewing. It is characterized by a unimodular matrix; the new loop bounds and new dependences are computed from this matrix. A 2d loop interchange has the transformation matrix $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$. These techniques are discussed in Chapter 2. Please refer to loop interchange for an example.

Internalization

A unimodular transformation specialized to exploit outer loop parallelism and locality for general loops. The technique is discussed in Chapter 2. The basic idea is to transform a loop in such way

that maximum number of dependences are not carried by the outer loop, and the size of the outer loop is maximum. The technique can be applied to increase dynamic locality as well.

Access Normalization

It is similar to unimodular loop transformations in that a matrix transformation is used. However, this matrix is derived from array access patterns, and need not be unimodular. The matrix is made invertible if needed. The objective is to simplify the array subscript expressions so as to match the simple data distributions and hence does not consider any other loop optimizations.

```
for i= p, N1-1, P
  for j= i, i+b-1
    for k= 0, N2-1
      B(i,j-i)= B(i,j-i)+ A(i,j+k)
    end for
  end for
end for
```

```
for u= p, b-1, P
  for v= u, u+N1+N2-2
    read A(*,v)
    for w= 0, N1-1
      B(w,u)= B(w,u)+ A(w,v)
    end for
  end for
end for
```

Note that addition of the line `read A(*,v)` is a *secondary* transformation.

Scaling

It is the converse of normalization in that it changes the loop stride from one to many. The transformation is characterized by an invertible integer matrix (generally not unimodular). Since most of the transformations operate on normalized loops, scaling is not generally employed. Scaling can provide more *natural looking* subscripts and bounds.

```
for i = 0, n/2                          for i = 0, n, 2
  for j = 0, n                          for j = 0, n
    A(2i,j) = A(2i-2,j)    ===>          A(i,j) = A(i-2,j)
  end for                               end for
end for                               end for
```

Rotation

Iterations in some dimensions can be shifted with respect to other dimensions so that the communication patterns become uniform. The transformed loop executes the iterations in a toroidal fashion. Although it does not change the shape of the iteration space, dependences are changed, and the transformation is not always legal.

```
for i = 0, n                          for i = 0, n
  for j = 0, n                          for j = 0, n
    A(i,j)=           ===>                A(i,(j-i) mod (n+1)) =
  end for                               end for
end for                               end for
```

The elements of A computed will be in the order $(0,0),(0,1),\ldots,(0,n),(1,n),(1,0),\ldots,(1,n-1),(2,n-1),(2,n),\ldots,(2,n-2),\ldots$ instead of the usual lexicographic order in the original loop.

Combing

This is equivalent to interchanging the loops of a strip mined loop. It is not always legal.

```
for i = 0, n                          for i = 0, 63
  A(i) = A(i-1)       ===>              for ii = i, n, 64
end for                                   A(i) = A(i-1)
                                        end for
                                      end for
```

Tiling

Tiling is also called blocking. It strips different loop levels to decrease the effective sizes of the inner loops so as to increase reuse. It can be viewed as increasing the grain size from an iteration to a collection of iterations (tile), where the outer loops step through the tiles and the inner loops step through the iterations in a tile. It is valid only if the loop levels to be tiled are fully permutable.

```
for i = 0, n                          for ii = 0, n, Si
  for j = 0, n                          for jj = 0, n, Sj
    ...               ====>               for i = ii, min (ii+Si-1,n)
```

```
  end for
end for
```

```
          for j = jj, min (jj+Sj-1,n)
                ...
          end for
        end for
      end for
    end for
```

A.3 Secondary Transformations

These are intended as improvements over transformed programs.

- Introducing communication primitives and changing them to more efficient primitives (like block transfers).
- Prefetching.
- Register binding, and
- Temporaries for index computation.

References

[1] W. Abu-Sufah. Improving the performance of virtual memory computers. Phd thesis, Univ. of Illinois at Urbana-Champaign, 1978.

[2] Randy Allen, David Callahan, and Ken Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Conference Record of the 14th Annual ACM Symposium on Principles of Programming Languages*, pages 63–76, Munich, West Germany, January 1987.

[3] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume 26, pages 39–50, Williamsburg, VA, April 1991.

[4] J. Anderson and M. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, volume 28, June 1993.

[5] D. Bacon, J. Chow, D. Ju, K. Muthukumar, and V. Sarkar. A compiler framework for restructuring data declarations to enhance cache and tlb effectiveness. In *Proceedings of CASCON 94*, Toronto, Canada, November 1994.

[6] D. Bacon, S. Graham, and O. Sharp. Compiler transformations for high-performance computing. *Computing Surveys*, 26(4):345–420, 1994.

[7] V. Balasundaram and Ken Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, volume 24, pages 41–53, Portland, OR, June 1989.

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

[9] Utpal Banerjee. A theory of loop permutations. In *Proceedings of Second Workshop on Programming Languag es and Compilers for Parallel Computing*, August 1989.

[10] Utpal Banerjee. Unimodular transformations of double loops. In *Proceedings of Third Workshop on Programming Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990.

[11] E. Barszcz and P. Frederickson. Nas multigrid benchmark. Technical report, NASA, Ames Research Center.

[12] Z. Chamski. Beyond convexity: Scanning 'non-convex polyhedra'. In 28th *Hawaii International Conference on System Sciences*, volume II: Software Technology, pages 73–82. IEEE, January 1995.

[13] S. Chatterjee, J.R. Gilbert, R. Schreiber, and S. Teng. Optimal evaluation of array expressions on massively parallel machines. *ACM Transactions on Programming Languages and Systems*, 17:123–156, 1995.

[14] Ron Cytron and Jeanne Ferrante. What's in a name? In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 19–27, 1987.

[15] M.L. Dowling. Optimum code parallelization using unimodular transformations. *Parallel Computing*, 16:155–171, 1990.

[16] P. Feautrier. Array expansion. In *Proceedings of the 1988 International Conference on Supercomputing*, 1988.

[17] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20, 1991.

[18] HPF Forum. Hpf: High performance fortran language specification. Technical report, HPF Forum, 1993.

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

[20] M.R. Garey and D.S. Johnson. *Computers and Intractability, A guide to the theory of NP-Completeness*. W.H. Freeman and Co., 1979.

[21] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An overview of the fortran d programming system. Technical Report CRPC-TR91121, Dept of computer Science, Rice University, 1991.

[22] W. Kelly and W. Pugh. A framework for unifying reordering transformations. Technical Report UMIACS-TR-92-126, University of Maryland, 1992.

[23] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. Technical Report UMIACS-TR-94-87, University of Maryland, 1994.

[24] P. M. W. Knijnenburg and A. J. C. Bik. On reducing overheads in loops. Technical Report 95–07, Department of Computer Science, Leiden University, The Netherlands, March 1995.

[25] K.B. Knobe. The subspace model: Shape-based compilation for parallel programs. Phd thesis, MIT, January 1997.

[26] D. Kulkarni, K.G. Kumar, A. Basu, and A. Paulraj. Loop partitioning for distributed memory multiprocessors as unimodular transformations. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[27] D. Kulkarni and M. Stumm. Computational alignment: A new, unified program transformation for local and global optimization. Technical Report CSRI-292, Computer Systems Research Institute, University of Toronto, January 1994.

[28] K.G. Kumar, D. Kulkarni, and A. Basu. Generalized unimodular loop transformations for distributed memory multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, Chicago, MI, July 1991.

[29] K.G. Kumar, D. Kulkarni, and A. Basu. Deriving good transformations for mapping nested loops on hierarchical parallel machines in polynomial time. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, July 1992.

[30] M. S. Lam. Software pipelining: an effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, volume 23, pages 318–328, Atlanta, GA, June 1988.

[31] L. Lamport. The parallel execution of do loops. *Communications of the ACM*, 17(2), 1974.

[32] C.H. Li. Program wanall. ftp ftp.cs.rice.edu, Rice University, 1992.

[33] Hui Li and Kenneth C. Sevcik. NUMACROS: Data Parallel Programming on NUMA Multiprocessors. In *Proceedings of Fourth Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, pages 247–263, September 1993.

[34] J. Li and M. Chen. The data alignment phase in compiling programs for distributed memory machines. *Journal of parallel and distributed computing*, 13:213–221, 1991.

[35] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. In *Proceedings of the Fifth Workshop on Programming Languages and Compilers for Parallel Computing*, August 1992.

[36] W. Li and K. Pingali. A singular loop transformation framework based on non -singular matrice. *International Journal of Parallel Programming*, 22(2), 1994.

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

[38] A.W. Lim and M.S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the 24th Annual ACM Symposium on Principles of Programming Languages*, January 1997.

[39] N. Manjikian and T. Abdelrahman. Array data layout for the reduction of cache conflicts. In *Proceedings of the 8th International Conference on Parallel and Distributed Computing Systems*, 1995.

[40] D.E. Maydan, J.L. Hennessy, and M.S. Lam. Efficient and exact data dependence analysis. *SIGPLAN Notices*, 26(6):1–14, 1991.

[41] C. Mosher and S. Hassanzadeh. Arco seismic benchmarks. Technical report, ARCO E&PT.

[42] M.F.P O'Boyle and G.A. Hedayat. Data alignment: Transformations to reduce communication on distributed memory architectures. In *Proceedings of the Scalable High Performance Computing Conference, IEE Press*, Williamsburg, 1992.

[43] D. Olshefski. Xcache601 and xcachers6k user's guide. Technical report, IBM T.J Watson Research Center. ftp:software.watson.ibm.com.

[44] F.P. Preparata and M.I. Shamos. Computational Geometry an Introduction. Springer-verlag, 1985.

[45] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, 1992.

[46] W. Pugh and D. Wonnacott. An exact method for analysis of value-based array data dependences. Technical Report CS-TR-3196, University of Maryland, 1993.

[47] J. Ramanujam. Non-singular transformations of nested loops. In *Supercomputing 92*, pages 214–223, 1992.

[48] V. Sarkar and R. Thekkath. A general framework for iteration-reordering loop transformations (technical summary). In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, volume 27, pages 175–187, San Francisco, CA, June 1992.

[49] A. Schrijver. *Theory of linear and integer programming*. Wiley, 1986.

[50] SPEC. Spec benchmarks. Technical report. Standard Performance Evaluation Corp.

[51] J. Torres and E. Ayguade. Partitioning the statement per iteration space using non-singular matrices. In *Proceedings of 1993 International Conference on Supercomputing, Tokyo, Japan, July 1993.*, 1993.

[52] J. Torres, E. Ayguade, J. Labarta, and M. Valero. Align and distribute-based linear loop transformations. In *Proceedings of Sixth Workshop on Programming Languages and Compilers for Parallel Computing*, 1993.

[53] P. Tu and D. Padua. Automatic array privatization. In *Proceedings of Sixth Workshop on Programming Languages and Compilers for Parallel Computing*, 1993.

[54] M. Wolf and M. Lam. A loop transformation theory and an algorithm to maxi mize parallelism. *IEEE Trans. Parallel Distributed Systems*, 2(4):452–471, 1991.

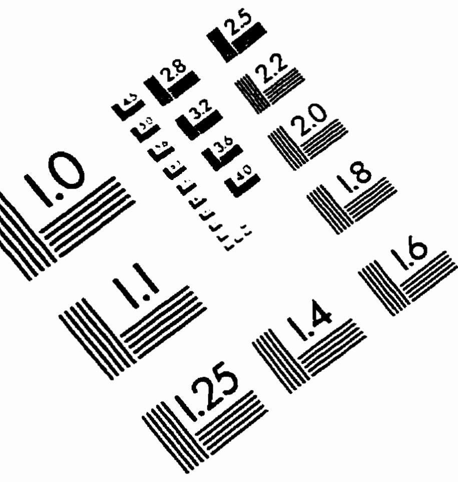
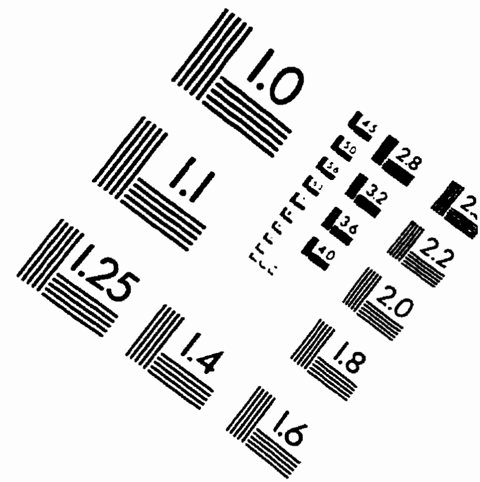
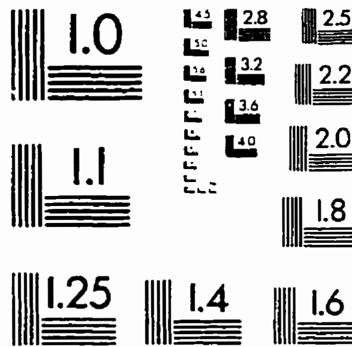
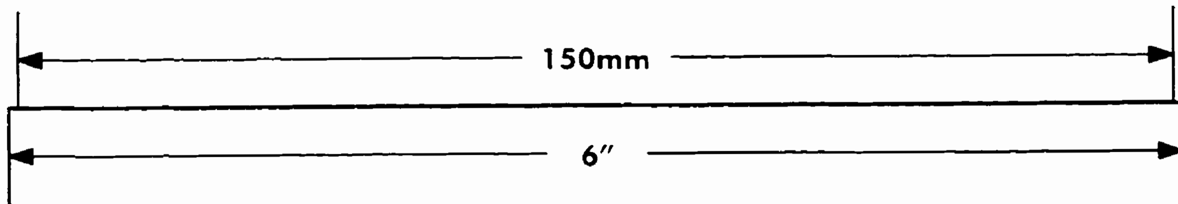
[55] M.E. Wolf and M.S. Lam. An algorithmic approach to compound loop transformation. In *Proceedings of Third Workshop on Programming Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990.

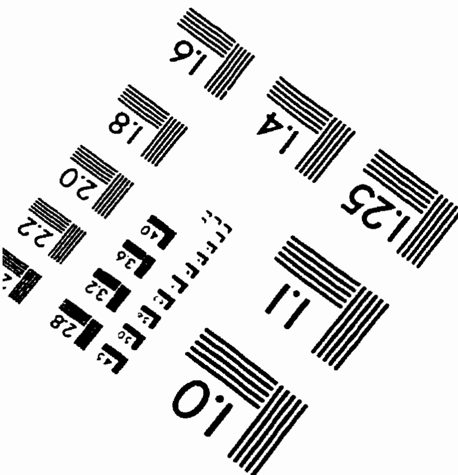
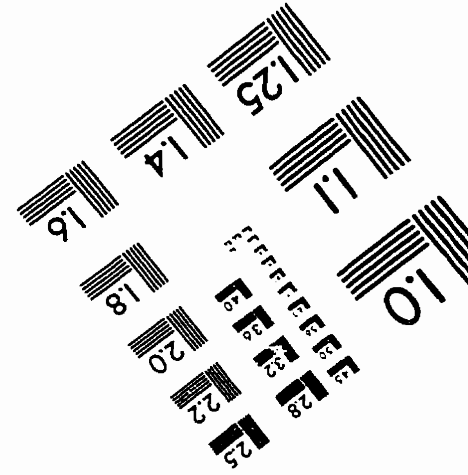
[56] Michael Wolfe. *Optimizing supercompilers for supercomputers*. The MIT Press, 1990.

[57] Michael Wolfe and Chau-Wen Tseng. The power test for data dependence. *IEEE Trans. Parallel Distributed Systems*, 3(5):591–601, 1992.

[58] H. Zima and B. Chapman. Supercompilers for parallel and vector computers. ACM Press, New York, NY, 1991.

IMAGE EVALUATION
TEST TARGET (QA-3)

APPLIED IMAGE . Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved