

REPRESENTING AND MINIMIZING MULTIDIMENSIONAL DEPENDENCIES

A Thesis

Presented to

The Graduate Faculty of The University of Akron

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

Krishna Chaitanya Chakilam

August, 2009

REPRESENTING AND MINIMIZING MULTIDIMENSIONAL DEPENDENCIES

Krishna Chaitanya Chakilam

Thesis

Approved:

Accepted:

---

Advisor

Dr. Timothy W. O'Neil

---

Dean of the College

Dr. Chand Midha

---

Faculty Reader

Dr. Kathy J. Liszka

---

Dean of the Graduate School

Dr. George R. Newkome

---

Faculty Reader

Dr. Timothy S. Margush

---

Date

---

Department Chair

Dr. Chien-Chung Chan

## ABSTRACT

Since processors spend most of the time in executing loop nests because of the dependencies, minimizing dependencies across loops is vital for compiler optimization. This paper explores two methods, sums of data dependencies and dominant data dependencies for eliminating dependencies in multi-dimensional loops. The first method eliminates dependencies by combining other dependencies and method two eliminates dependencies by making use of the distance of the dependence. We also present how to apply loop transformation techniques to these methods to explore much better results. To conclude we provide an algorithm for minimizing the number of dependencies in multi-dimensional loops.

## TABLE OF CONTENTS

	Page
LIST OF FIGURES .....	vi
LIST OF ALGORITHMS.....	vii
CHAPTER	
I. INTRODUCTION .....	1
1.1 Research Objective .....	4
1.2 Organization.....	5
II. BACKGROUND.....	6
2.1 Read-After-Write (RAW) .....	6
2.2 Write-After-Read (WAR) .....	7
2.3 Write-After-Write (WAW).....	7
2.4 Restrictions of the Model.....	8
2.5 Intra and Inter Iteration dependencies.....	8
2.6 Characteristic Value and Subsumption.....	10
2.7 The Translation Property of Data Dependency .....	10
2.8 Loop Transformation .....	11
2.9 Conclusion .....	12
III. MULTI-DIMENSIONAL DEPENDENCIES .....	13

3.1	Dependence Notation in Multidimensional Loops .....	13
3.2	The Translation Property of Data Dependencies in 2-Dimensional Loops.....	14
3.3	Negative Coordinates.....	15
3.4	Subsumption in Negative Coordinates.....	15
3.5	Sums of Data Dependencies in Multiple Dimensional Loops .....	16
3.6	Dominant Data Dependencies in Multidimensional Loops .....	18
3.7	Algorithm for Eliminating Redundant Data Dependencies in Multidimensional Loops .....	19
3.8	Example to Demonstrate the Algorithm .....	25
3.9	Conclusion .....	27
IV. ELIMINATION OF DEPENDENCIES USING LOOP TRANSFORMATION METHODS .....		29
4.1	Vertical Shearing (Loop Skewing) .....	29
4.2	Horizontal Shearing .....	31
4.3	Wavefronting .....	33
4.4	Combining Loop Skewing with Data Dependence Elimination.....	33
4.5	Combining Horizontal Skewing with Data Dependence Elimination .....	38
4.6	Combining Wavefronting with Data Dependence Elimination .....	40
4.7	Conclusion .....	41
V. CONCLUSIONS AND FUTURE WORK .....		43
5.1	Contributions.....	43
5.2	Future Work .....	44
REFERENCES .....		46

## LIST OF FIGURES

Figure	Page
2.1. Loop Carried Dependencies (a) Example code with loop-carried dependencies; (b) Its parallel schedule .....	9
3.1 Example two-dimensional loop having dependence at distance (2,3).....	14
3.2 Sample code representing loop carried dependencies in two dimensional loop.....	25
3.3 Scheduling of the example code .....	27
4.1 Iteration space after vertical shearing .....	30
4.2 Iteration space after horizontal shearing .....	32
4.3 Sample code with loop carried dependencies .....	34
4.4 Sample code with loop carried dependencies after loop skewing with skew factor, $f = 1$ .....	35
4.5 Sample code with loop carried dependencies after horizontal skewing with skew factor $f = 1$ .....	39
4.6 Sample code with loop carried dependencies after wavefronting .....	40

## LIST OF ALGORITHMS

Algorithm	Page
3.1 Algorithm for finding maximum characteristic value .....	21
3.2 Algorithm for eliminating dependencies in same outer loop distance.....	22
3.3 Algorithm for finding Dominant Dependencies (step 1) .....	23
3.4 Algorithm for finding Dominant Dependencies (step 2) .....	24

## CHAPTER I

### INTRODUCTION

Dependencies in programs block the compiler from providing high throughput. They reduce the amount of parallelism that can be extracted from programs by causing memory access problems. The dependencies in multidimensional loops get more complex with multiple index variable (MIV) array subscripts. The loop-carried dependencies in multidimensional loops cause more problems, increasing the number of halts in the pipeline. Thus, identifying and eliminating dependencies in multidimensional loops is essential. There are many optimization techniques available to allow a compiler to deal with these problems.

There has been a lot of research on minimization of loop-carried dependencies in multidimensional loops. Most of the work introduces different transformation techniques such as loop unrolling, loop splitting, loop fusion, loop skewing and many others. We will briefly discuss some of these contributions which are similar to our work.

The most successful work in this domain has been loop skewing. The authors in [1] clearly explain this concept. This paper discuss a new implementation of the wavefront transformation called loop skewing, which is a simple manipulation of loop bounds and is combined with loop interchanging to produce wavefronting. This method

basically eliminates dependencies by changing the shape of the iteration space, i.e. it changes the order of array access.

A loop parallelizing method using horizontal and vertical shearing is described in [2]. The article demonstrates the advantages of horizontal shearing, which is still an area new and largely unexplored topic. It also discusses advantages and disadvantages of both shearing techniques and suggests how a compiler should select a particular technique for generating parallelized code. The article also discusses findings where the code consists of a group of loop bodies which can execute in parallel. It also derives the relationship between data dependence, shearing angle and blocking for SIMD optimization.

The authors in [3] propose a new technique to increase parallelism in nested loops called loop striping. This technique, unlike other loop transformation methods like wavefront methods, does not change the execution sequence and order of array access, minimizing the complicated loop bound calculations. Loop striping separates iterations into stripes where a stripe is a group of iterations which are independent of each other.

A new approach for combining all loop transformation methods like loop reversal, skewing, and loop interchange is explained in [4]. The authors claim that existing compilers must choose appropriate transformations after each step for vectorizing and parallelizing code, which is inadequate because the choice and selection of optimizations are mostly program dependent and cannot be evaluated after each step. The article presents a new approach based on matrix transformations. The article also presents an efficient loop transformation algorithm based on this approach to maximize parallelism in a loop.

In [5] the authors discuss the parallelization of flow and anti-dependence loops with non-uniform dependencies. They mainly focus on an improved region partitioning method for minimizing the size of sequential regions and maximizing parallelism. Their approach is based on convex hull theory which has information to handle non-uniform dependencies.

As many of the loop transformations affect cache behavior, instruction scheduling and register allocation of a machine, the authors in [6] discuss eliminating the interdependence of loop transformations on machine characteristics. The paper presents a method to estimate machine cycle time considering cache misses, software pipelining, etc., and then develops an algorithm that will select a set of transformations having high overall performance.

In [7] the authors provide a new method of dependence elimination by applying different loop transformation methods to different statements of the loop body instead of applying a particular transformation to the whole loop at once. It is considered as alignment before transformation. The extension to this can be seen in [8] which further exploits coarse-grain parallelism on a MIMD system by including a new step between the alignment and transformation which is based on the remainder transformation[9].

In [10] authors provide a compiler strategy for estimating the cost of executing a given loop nest in terms of the number of cache line references. The optimization algorithm takes advantage of the case where the cache miss rate for a program is high and tries to improve data locality. The algorithm makes use of compound loop transformations like loop permutation, fusion and distribution. The paper also presents

empirical results for kernel and benchmark programs that validate the effectiveness of the strategy.

Thus we can conclude that there has been a lot of work already done in the field of compiler optimization, based on several loop transformation techniques. In this research, we are going to minimize the dependencies in code by using the loop skewing transformation and then applying an algorithm we develop which further tries to minimize the dependencies.

### 1.1 Research Objective

1. Our overall goal is to develop a method for eliminating dependencies in multi-dimensional loops but we focus on two dimensional loops in this thesis.
2. We develop a representation of data dependencies in multidimensional loops and generate an algorithm for eliminating redundant data dependencies in multidimensional loops, extending work done in [7].
3. We apply loop transformation, namely loop skewing, to the code. To the skewed loop we again apply the algorithm we generated to further minimize dependencies.

## 1.2 Organization

Chapter 2 discusses the basic concepts of data dependencies, loop transformation, etc., which lay the foundation for this thesis. It also explains new terminology such as subsumption, characteristic value, sums of dependencies, and dominant data dependence which are used in developing a redundant dependence elimination algorithm.

Chapter 3 presents a new representation for dependencies in multidimensional loops, generates proof for sums of dependencies in two dimensional loops and provides an algorithm for minimizing dependencies.

In Chapter 4 we first apply loop transformation algorithms to code, and then eliminate the dependencies by repeating the same algorithms used in chapter 3 and analyze the results produced.

Finally Chapter 5 we draw conclusions from our work and provide recommendations for future work.

Note: The preliminary version of the work in this thesis will be published [13].

## CHAPTER II

### BACKGROUND

In this chapter we discuss the basic concepts of dependencies and review the terminology used for studying different properties and characteristics of them. As the primary focus of this paper is to eliminate data dependencies in multi-dimensional loops we will discuss terminology of data dependencies related to two dimensional loops. Since dependencies in code stop a compiler from performing at its peak efficiency we start by analyzing the types of dependencies and hazards caused by them.

Consider two instructions  $i$  and  $j$ , with  $i$  executing before  $j$ . There are basically four different types of dependencies[11].

#### 2.1 Read-After-Write (RAW)

This is the most common kind of hazard. RAW hazard exists when instruction  $j$  tries to read a location before instruction  $i$  writes to it. In other words there exists a RAW hazard between  $i$  and  $j$  if  $j$  tries to read before  $i$  completes its writeback stage of the pipeline. This hazard is sometimes referred to as a *true dependence*.

## 2.2 Write-After-Read (WAR)

This kind of hazard exists when instruction  $j$  tries to write to a memory location before it is read by instruction  $i$  which causes instruction  $i$  to read a wrong value. These hazards can be easily eliminated by using different registers for two instructions. This hazard is sometimes referred to as an *anti-dependence*.

## 2.3 Write-After-Write (WAW)

This kind of hazard exists when instruction  $j$  tries to write to a memory location before it is written by instruction  $i$ . Since  $i$  is supposed to execute before  $j$  the memory location gets the wrong value stored into it. This can be eliminated if the serial order execution of instructions is maintained.

The WAR and WAW hazards arise only because two instructions try to make use of the same memory location at the same time. If the register names are changed so that conflict doesn't happen these hazards can be eliminated. This is not the case in true dependencies because in RAW hazards there is a transfer of data from one register to another so this hazard cannot be eliminated by changing the names of the registers. Therefore, we mainly concentrate on eliminating this kind of dependence.

## 2.4 Restrictions of the Model

In this section we point out the restrictions placed on the model we are developing.

- i. The array subscripts which are linear subscripts of the loop induction variables have coefficient as only one.
- ii. Single induction variable (SIV) subscripts, meaning that only one index occurs in a given subscript position; by necessity this limits the study to arrays having at most the same dimension as the number of loops in the nest.
- iii. Upper and lower bounds of loops in the nest are independent of the other loop induction variables.
- iv. Instructions within the loop nest execute serially; we can develop separate methods later to bring out intra-iteration parallelism.
- v. Normalized loops starting at 1 with stride 1.

These assumptions made here are consistent with other work on the topic [14].

## 2.5 Intra and Inter Iteration dependencies

If a dependence exists between two instructions of the same iteration then it is called an *intra-iteration* dependence. If a dependence exists between instruction A of iteration  $x$  and instruction B of iteration  $y$  when  $x \neq y$  then it is called an *inter-iteration* or *loop-carried* dependence. The *distance* of the dependence is defined as the difference between  $x$  and  $y$ . Therefore, a loop-carried dependence has distance greater than zero and

the distance of an intra-iteration dependence is zero. We will refer to a dependence from instruction A to instruction B having distance d using the notation  $(A \rightarrow B, d)$ . For now we assume a single loop as in our previous work, so the distance is an integer rather than a vector.

Consider the following example code in Fig. 2.1 and we will see more clearly the kinds of dependencies that exist in code. By unrolling the loop we can see there are 8 RAW hazards in the code. They are

$$(1 \rightarrow 2, 0), (1 \rightarrow 4, 0), (2 \rightarrow 3, 1), (2 \rightarrow 5, 1), (3 \rightarrow 2, 1), (3 \rightarrow 4, 1), (4 \rightarrow 3, 2), (4 \rightarrow 5, 2).$$

We can eliminate all of these dependencies except  $(3 \rightarrow 2, 1)$  by applying the elimination algorithm from [12]. This is the only restriction we need to obey when constructing the loop's repeating parallel schedule, as shown in Fig. 2.1(b).

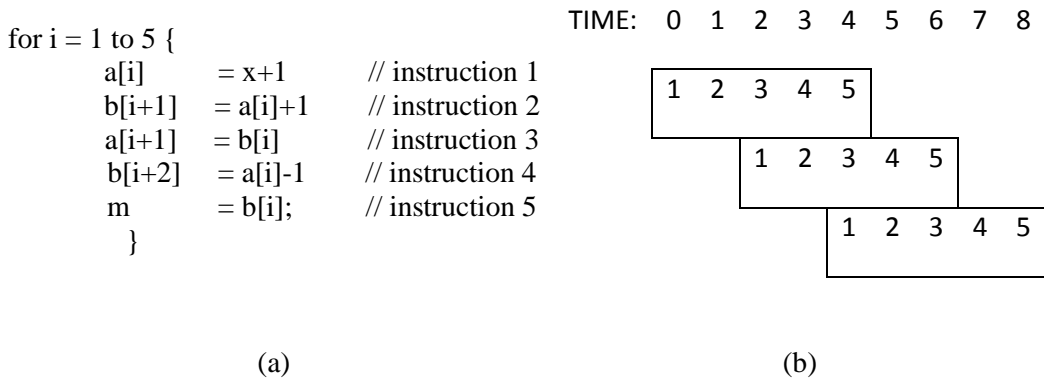


Figure 2.1: Loop Carried Dependencies (a) Example code with loop-carried dependencies; (b) Its parallel schedule.

## 2.6 Characteristic Value and Subsumption

As seen above, when attempting to increase the parallelism in the code, not all dependencies need to be considered. Some concepts like subsumption and characteristic data dependencies, which eliminate data dependencies in code, are discussed in depth in [12]. By implementing these ideas in the elimination algorithm from [12] we can greatly reduce the number of data dependencies.

Let us consider a data dependence  $(X \rightarrow Y, d)$ . If  $X$  executes at time  $T_x$  and  $Y$  executes at time  $T_y$  relative to the current iteration then the *characteristic value* of the data dependence is defined as difference in relative start times within one iteration of the two instructions, i.e.  $T_x - T_y$ . For example, in the dependence  $(2 \rightarrow 5, 1)$  in Figure 2.1(a), the characteristic value is -3. (For simplicity assume that all instructions take one time unit to complete).

Finally, this dependence is *fulfilled* or satisfied if  $T_x < T_y$  or if  $X$  executes before  $Y$ . Whenever fulfillment of dependence  $A$  automatically satisfies dependence  $B$  then we say that  $A$  *subsumes*  $B$ , denoted as  $A \supseteq B$ . For example, in Fig. 2.1(a) we can see that  $(2 \rightarrow 3, 1) \supseteq (2 \rightarrow 5, 1)$ . In the same example note that  $(3 \rightarrow 2, 1) \supseteq (4 \rightarrow 3, 1)$  as well by this definition. The importance of this will be discussed next.

## 2.7 The Translation Property of Data Dependency

Based on this notation the translation property of data dependencies was derived in [12].

Theorem 2.1: Let  $d$  be an iteration distance, and let  $A: (I_i \rightarrow I_j, d)$  and  $B: (I_m \rightarrow I_n, d)$  be two data dependencies. Then  $A$  and  $B$  represent the same dependence

if  $i - j = m - n$ . QED

Proof: Because of this translation property, the dependency relation  $(I_i \rightarrow I_j, d)$  is the same as  $(I_{i-j} \rightarrow I_0, d)$  if  $i \geq j$  and  $(I_0 \rightarrow I_{j-i}, d)$  otherwise, so we need only keep track of the characteristic value of a data dependence. Because of this, we will henceforth refer to the dependency  $A: (I_i \rightarrow I_j, d)$  as  $A: (\lambda_A, d)$  where  $\lambda_A = i - j$ . We can also say that two dependences are *equal* if their characteristic values are equal.

## 2.8 Loop Transformation

Many loop-carried dependencies can be eliminated by applying a series of loop transformation algorithms. These transformations change the order of array access by exchanging the outer loop with the inner, breaking a loop into multiple loops and using many other methods. Some loop transformation techniques are loop interchange, loop splitting, loop inversion, loop fusion, and loop skewing. Some of these transformations include changing the array index bounds. Each transformation should be checked for validity after transforming. A transformation is said to be valid if it preserves the sequence of all dependencies after transforming. A transformation may not be beneficial by itself but it may help reduce dependencies when combined with others.

## 2.9 Conclusion

In this chapter we discussed various types of dependencies that can exist in the code. We also discussed terminology and notations used to represent dependencies. We also explained what exactly is loop transformation, various types of transformations and characteristic features of them. In the next chapter we will discuss more on applying loop transformations to the elimination algorithm which is the main focus of this research.

## CHAPTER III

### MULTI-DIMENSIONAL DEPENDENCIES

This chapter introduces a new type of notation for the dependencies in multidimensional loops, basic terminology which is specific for multidimensional loops and with supporting proofs. We also discuss the techniques for eliminating redundant dependencies. The proposed algorithms are explained with the help of an example.

#### 3.1 Dependence Notation in Multidimensional Loops

Unlike the one-dimensional case, the notation for dependencies in multidimensional loops cannot be represented in the form of  $(\lambda, d)$  because distance in a multidimensional loop is not just a scalar variable. Instead it consists of distances across different loops and thus is represented in the form of a tuple. The notation used in this paper will be of the form  $(\lambda, (a, b, \dots, n))$  where ' $\lambda$ ' represents the characteristic value and  $(a, b, \dots, n)$  represents the distance across each loop of the dependence. For example, the true dependence in Figure 3.1 is represented as  $(1, (2, 3))$  which should be interpreted as a dependence with a characteristic value of 1 and a distance of 2 iterations of the first loop and 3 of the second loop.

```

for x = 1 to 5
{
  for y = 1 to 5
  {
    a[x+2][y+3] = b[1][1];
    c[x+1][y+1] = a[x][y];
  }
}

```

Figure 3.1: Example two-dimensional loop having dependence at distance (2,3).

As seen from the Figure 3.1 above the second instruction has dependence with the first instruction at a distance of 2 in the x-loop and 3 in the y-loop. The result computed on iteration (1,1) is used on iteration (3,4).

### 3.2 The Translation Property of Data Dependencies in 2-Dimensional Loops

For the remainder of the thesis, we will assume 2-dimensional loops for simplicity with the extension to more dimensions as future work. As we progress, it will be useful to note that the 2-dimensional dependence distance  $(x,y)$  corresponds to a 1-dimensional distance of  $x*n+y$ , where  $n$  is the number of iterations of the inner loop. This can be seen by realizing that, starting from the dependence source, one jumps over  $x$  occurrences of the outer loop (i.e.  $x*n$  total iterations), plus an additional  $y$  iterations of the inner loop, to arrive at the dependence sink. Considering these ideas we now prove the translation property of 2-dimensional loops.

Theorem 3.1: Let  $(d_x,d_y)$  be an iteration distance, and let  $A: (I_i \rightarrow I_j, (d_x,d_y))$  and  $B: (I_m \rightarrow I_n, (d_x,d_y))$  be two data dependencies. Then  $A$  and  $B$  represent the same dependence if  $i - j = m - n$ .

Proof: Without loss of generality assume that  $i \leq j$  and  $m \leq n$ . The dependency relation  $(I_i \rightarrow I_j, (d_x, d_y))$  is the same as  $(I_0 \rightarrow I_{j-i}, d_x * k + d_y)$  where  $k$  is number of iterations of inner loop. Similarly,  $(I_m \rightarrow I_n, (d_x, d_y))$  is the same as  $(I_0 \rightarrow I_{m-n}, d_x * k + d_y)$ . By Theorem 2.4 A and B are the same if  $i - j = m - n$ . QED.

Because of this, we will henceforth refer to the dependency A:  $(I_i \rightarrow I_j, (d_x, d_y))$  as A:  $(\lambda_A, (d_x, d_y))$  where  $\lambda_A = i - j$ . We can also say that two dependences are equal if their characteristic values are equal.

### 3.3 Negative Coordinates

The first non-zero coordinate in the distance tuple must be positive for the dependence to exist at all [14]. However, later coordinates in the distance tuple may not always be positive. They may also be negative. For example, consider the dependence  $(1, (2, -3))$ . This should be interpreted as a dependence occurring 3 iterations of the inner loop before the start of the second iteration of the outer loop. The dependence with a negative inner loop co-ordinate subsumes all other dependencies of the same outer loop co-ordinate. For example, the dependence  $(1, (2, -2))$  subsumes dependencies  $(2, (2, -1))$  and  $(1, (2, 1))$ . This is formally proven next.

### 3.4 Subsumption in Negative Coordinates

With our previous ideas in mind we prove that the dependence having the smallest inner loop distance subsumes all dependencies of the same outer loop distance.

Lemma 3.1: Consider the data dependencies A:  $(a, (x, y_a))$  and B:  $(b, (x, y_b))$  with  $a < b$  and  $y_b < y_a$ . Then  $B \supseteq A$ .

Proof: By the Translation Property, the dependence A can be viewed as representing a data dependence between instruction  $I_{a+1}$  of iteration 0 and  $I_1$  of iteration  $x*n+y_a$  where  $n$  is number of iterations of the inner loop. Likewise, the dependence B can be viewed as representing a data dependence between instruction  $I_{b+1}$  of iteration 0 and  $I_1$  of iteration  $x*n+y_b$ . Because we assume sequential execution within iterations and  $a < b$ , instruction  $I_{a+1}$  of iteration 0 must complete execution before instruction  $I_{b+1}$  of the same iteration can begin. Similarly, since  $y_b < y_a$ , instruction  $I_1$  iteration  $x*n+y_b$  must precede instruction  $I_1$  of iteration  $x*n+y_a$ . Therefore, satisfying the dependence B automatically satisfies dependence A which logically means dependence B subsumes dependence A. QED

Thus the dependence having the smallest inner loop distance and largest characteristic value subsumes all dependencies of the same outer loop distance. This permits us to eliminate some dependencies having the same inner distance.

### 3.5 Sums of Data Dependencies in Multiple Dimensional Loops

In addition to basic subsumption seen above, if a dependence can be subsumed by some combination of other dependencies, it should also be removed. For example, in Figure 2.1, the concatenation of the dependencies  $(2 \rightarrow 3, 1)$ ,  $(3 \rightarrow 4, 1)$  and  $(4 \rightarrow 5, 1)$  yields the dependence  $(2 \rightarrow 5, 1)$ , making its explicit consideration unnecessary. We will abuse the traditional language and refer to this process as *adding* the dependencies, so that the

*sum* of the dependencies in question results. Following our prior notation we prove the following theorem.

Theorem 3.2: Given two 2-dimensional data dependencies  $(\lambda_a, (d_{ax}, d_{ay}))$  and  $(\lambda_b, (d_{bx}, d_{by}))$ , their sum, written  $(\lambda_a, (d_{ax}, d_{ay})) + (\lambda_b, (d_{bx}, d_{by}))$ , is equal to the dependence

$$(\lambda_a + \lambda_b + 1, (d_{ax} + d_{bx} - 1, d_{ay} + d_{by})).$$

Proof: By the Translation Property from [12] the first ordered pair represents data dependence between instructions  $I_{a+1}$  of iteration 0 and  $I_1$  of iteration  $(d_{ax}, d_{ay})$ . Assuming all the instructions are executed in order,  $I_{a+1}$  of iteration 0 will begin at time  $(\lambda_{ax} - 1) * n + \lambda_{ay}$  and so instruction  $I_1$  of iteration  $(d_{ax}, d_{ay})$  cannot start sooner than  $((\lambda_{ax} - 1) * n + \lambda_{ay}) + 1$ . Similarly, the second ordered pair represents dependence between instructions  $I_{b+1}$  of iteration  $(d_{ax}, d_{ay})$  and  $I_1$  of iteration  $(d_{ax} + d_{bx}, d_{ay} + d_{by})$ . Therefore,  $I_{b+1}$  begins at time  $((\lambda_{ax} - 1) * n + \lambda_{ay}) + 1 + ((\lambda_{bx} - 1) * n + \lambda_{by})$  and  $I_1$  of iteration  $(d_{ax} + d_{bx}, d_{ay} + d_{by})$  begins at time  $(\lambda_{ax} + \lambda_{bx} - 2) * n + (\lambda_{ay} + \lambda_{by} + 2)$ . Since instruction  $I_{a+b+2}$  of iteration 0 begins at  $[(\lambda_{ax} - 1) * n + \lambda_{ay}] + [(\lambda_{bx} - 1) * n + \lambda_{by}] + 1$ , the sum is equal to a data dependence between this and  $I_1$  of iteration  $(d_{ax} + d_{bx}, d_{ay} + d_{by})$  which is represented as

$$(\lambda_a + \lambda_b + 1, (d_{ax} + d_{bx} - 2) * n + (d_{ay} + d_{by})).$$

Changing back to our previous notation the equation becomes

$$(\lambda_a + \lambda_b + 1, (d_{ax} + d_{bx} - 1), (d_{ay} + d_{by}))$$

since

$$(\lambda_a + \lambda_b + 1, (d_{ax} + d_{bx} - 2) * n + (d_{ay} + d_{by}))$$

$$= (\lambda_a + \lambda_b + 1, ((d_{ax} + d_{bx} - 1) - 1) * n + (d_{ay} + d_{by}))$$

$$= (\lambda_a + \lambda_b + 1, ((d_{ax} + d_{bx} - 1), (d_{ay} + d_{by})))$$

Therefore, the sum of dependencies in a multi-dimensional loop is

$$(\lambda_a, (d_{ax}, d_{ay})) + (\lambda_b, (d_{bx}, d_{by})) = (\lambda_a + \lambda_b + 1, (d_{ax} + d_{bx} - 1, d_{ay} + d_{by})). \quad \text{QED}$$

The sum can be inductively showed as

$$\sum_{i=1}^n Ai = ((\sum_{i=1}^n 1) - 1, (\sum_{i=1}^n -1) + 1, \sum_{i=1}^n)_{yi}$$

If we consider data dependence  $D = (\lambda, (d_x, d_y))$  and  $p$  as any positive integer then we can generalize above formula as  $p \cdot D = (p(\lambda + 1) - 1, ((p(d_x - 1)) + 1, pd_y))$ .

### 3.6 Dominant Data Dependencies in Multidimensional Loops

The sums of data dependencies discussed in the last chapter demonstrate how an individual dependence  $\Lambda$  can subsume all other dependencies of the same distance. This section deals with the same concept, i.e. reducing data dependencies through a different method. The *dominant data dependence*, denoted as  $\Delta$ , eliminates all dependencies for a certain distance unlike sums of dependencies which only reduce the total number of dependencies to one. For a given set of data dependencies there is a possibility that the sum of characteristic values of a subset of dependencies with varying distances can

subsume another dependency if the sum of distances for the subset is equal to the distance of the subsumed dependency.

In two dimensional loops, even though the distance tuple has outer loop coordinate and an inner loop coordinate, the equality of the sum of distances of the outer loop for the subset of dependencies and the outer loop distance for the subsumed dependency is enough. Equaling the inner loop distance is not required but the inner loop distance of the subsumed dependence should be less than or equal to the inner loop distance of the subset of dependencies because the dependency with the least inner loop distance subsumes all other dependencies which is proved in Section 3.2.

### 3.7 Algorithm for Eliminating Redundant Data Dependencies in Multidimensional Loops

The following algorithm takes a set of data dependencies  $S$ , maximum iteration distance  $M$ , least second coordinate (a) and highest second coordinate (b) as input and gives as output a set of dependencies with all the redundant dependencies removed.

The module in Algorithm 3.1 finds the maximum characteristic value for each distance and deletes all the dependencies which have characteristic value less than the maximum.

The module in Algorithm 3.2 is for eliminating dependencies in the same distance, i.e. outer loop distance. There is a possibility that the dependencies with the

same outer loop distances but with different inner loop distances can be reduced by applying Lemma 3.1.

In the module in Algorithm 3.3 and 3.4 we find the largest sum of characteristic values whose outer loop distance adds to  $d_x$  and inner loop distance adds to  $d_y$ . The dependence is eliminated if this sum is greater than the characteristic value of the dependence in which case this new characteristic value becomes the dominant data dependence for the distance  $d$ . This task is achieved in two parts. In the first part we keep the possible dependencies which can eliminate the dependency  $(\lambda, (d_i, d_y))$  in variable  $P_i$  i.e. if the first coordinates of both the dependencies sum up to the first coordinate of the dependency  $(\lambda, (d_i, d_y))$ . In the second part we check whether the second coordinates for the dependencies in  $P_i$  sum up to the second coordinate of the dependency  $(\lambda, (d_i, d_y))$  if this happens we delete the dependency  $(\lambda, (d_i, d_y))$  and dominant data dependency for this distance  $(d_i, d_y)$  is the sum of the characteristic values of the dependencies in  $P_i$ .

```
/* Step 1 */
```

Input: List of dependencies in sample code.

Output: List of dependencies with only maximum characteristic value for a distance.

```
for i = 1 to M do
  for j = a to b do
     $\Lambda[i,j] \leftarrow -\infty$ 
    for all data dependencies in  $(\lambda, (d_i, d_j))$  do
      if  $\lambda > \Lambda[i,j]$  then
        if  $\Lambda[i,j] > -\infty$  then
          delete data dependence  $(\Lambda[i,j], (d_i, d_j))$  from S
        end if
         $\Lambda[i,j] \leftarrow \lambda$ 
      else
        delete the data dependence  $(\lambda, (d_i, d_j))$  from S
      end if
    end for
  end for
end for
```

Algorithm 3.1: Algorithm for finding maximum characteristic value.

```
/* Step 2 */
```

Input: List of dependencies in sample after finding maximum characteristic value.

Output: List of dependencies having largest characteristic value and minimum inner loop iteration distance.

```
for i = 1 to D(total number of dependencies)
  for j = a to b
    n ← ∞
    v ← -∞
    for all dependencies with outer loop index i
      if  $d_y < n$  then
        if  $\lambda > v$  then
          if  $n < \infty \ \&\& \ v > -\infty$  then
            delete data dependence (v, (i,n))
          end if
          n ←  $d_y$ 
          v ←  $\lambda$ 
        end if
      else
        delete data dependence ( $\lambda$ , (i,  $d_y$ ))
      end if
    end for
  end for
end for
```

Algorithm 3.2: Algorithm for eliminating dependencies in same outer loop distance.

```
/* Step 3 */
```

Input: Dependency list after finding minimum inner loop iteration distance.

Output: Dependency list which can be eliminated using dominant dependence.

```
 $\Delta[1,a] = \Lambda[1,a]$ 
```

```
for i = 2 to D do
```

```
  M  $\leftarrow$   $-\infty$ 
```

```
  for j = 1 to i/2 do
```

```
    if  $(\lambda_j)$  in S
```

```
      if M <  $\Delta[j] + \Delta[i-j] + 1$  then
```

```
        M =  $\Delta[j] + \Delta[i-j] + 1$ 
```

```
         $P_{\lambda_i} = ((\lambda_1, (d_j, d_{y1})) + (\lambda_2, (d_{i-j}, d_{y2})))$ 
```

```
         $P_{x_i} = d_j + d_{i-j} - 1$ 
```

```
         $P_{y_i} = d_{y1} + d_{y2}$ 
```

```
      end if
```

```
    end if
```

```
  end for
```

```
end for
```

Algorithm 3.3: Algorithm for finding Dominant Dependencies (step 1).

Input: Probable dependence list which can be eliminated using dominant dependence.

Output: Gives the final dependencies list that exist after finding dominant dependencies.

```
for i=2 to D do
  if  $(d_i, d_y) = (P_{x_i}, P_{y_i})$  and  $\lambda_i < P_{\lambda_i}$  then
     $\Delta[i] = P_{\lambda_i}$ 
    delete  $(\Lambda[i], (d_i, d_y))$  from S
  else
     $\Delta[i] \leftarrow \Lambda[i]$ 
  end if
end for
```

Algorithm 3.4: Algorithm for finding Dominant Dependencies (step 2).

### 3.8 Example to Demonstrate the Algorithm

The algorithm is demonstrated for the sample program in Figure 3.2 below.

```
for i = 0 to 4
{
for j = 0 to 4
{
a[i+1][j+1] = a[1][1] /* instruction 1*/
b[i+4][j+2] = a[i][j+5] + a[i-1][j] /* instruction 2*/
a[i+3][j-2] = a[i+3][j-3] /* instruction 3*/
a[i-3][j-3] = a[i][j+1] + b[i+3][j] /* instruction 4*/
c[i+1][j+1] = a[i][j+1] /* instruction 5*/
c[i][j] = b[i+1][j+3] + b[i][j+4] /* instruction 6*/
}
}
}
```

Figure 3.2: Sample code representing loop carried dependencies in two dimensional loop.

The following is the list of dependencies  $S$  for the above example.

In this example code we found 11 dependencies which are  $(1 \rightarrow 2, (1, -4))$ ,  $(1 \rightarrow 4, (1, 0))$ ,  $(1 \rightarrow 5, (1, 0))$ ,  $(1 \rightarrow 2, (2, 1))$ ,  $(3 \rightarrow 2, (3, -7))$ ,  $(3 \rightarrow 2, (4, -2))$ ,  $((2 \rightarrow 4, (1, 2))$ ,  $(3 \rightarrow 4, (3, -3))$ ,  $(3 \rightarrow 5, (3, -3))$ ,  $(2 \rightarrow 6, (3, -1))$ , and  $(2 \rightarrow 6, (4, -2))$ . By following Theorem 2.1 we can represent these dependencies as

$$S = \{(-1, (1, -4)), (-3, (1, 0)), (-4, (1, 0)), (-2, (1, 2)), (1, (3, -7)), (1, (4, -2)), (-1, (2, 1)), (-1, (3, -3)), (-4, (3, -1)), (-2, (3, -3)), (-4, (4, -2))\}$$

In step 1 we keep the dependencies with the maximum characteristic value for a distance and delete all other dependencies for that distance. Therefore, the dependence

$(-4,(1,0))$  and  $(-2,(3,-3))$  is deleted since the characteristic value of this dependency is less than  $(-3,(1,0))$  and  $(-1,(3,-3))$  respectively.

Thus, the dependence list S after finding the maximum characteristic data dependence is

$$S = \{(-1,(1,-4)), (-3,(1,0)), (-2,(1,2)), (-1,(2,1)), (-4,(3,-1)), (1,(3,-7)), (-1,(3,-3)), (1,(4,-2)), (-4,(4,-2))\}$$

In step 2 the dependence with largest characteristic value and smallest inner loop distance value is stored for same outer loop distance and the remaining dependencies are deleted. Therefore, the dependency list S at the end of step 2 will be

$$S = \{(-1,(1,-4)), (-1,(2,1)), (1,(3,-7)), (1,(4,-2))\}$$

In step 3 ,  $\Delta[1,a]$  is set to  $\Lambda[1,a]$  because the dominant data dependence for this dependency can be only  $\Lambda[1,a]$  since the outer loop distance 1 cannot be further decomposed into other dependencies. For all the other dependencies for which the outer loop coordinate is greater than 2 we apply the algorithm.

In the next for loop, the coordinates of the dependencies are compared with the  $P_{xi}$ , and  $P_{yi}$  values. If the characteristic value of the dependency is less than  $P_{\lambda_i}$  then the dominant data dependence for that distance is set to  $P_{yi}$  and the dependence is deleted. If the comparison evaluates to false, the dominant data dependence for the distance will be set to the characteristic value of the distance and the dependence is not deleted.

Therefore, when  $I=4$  in this for loop the comparison evaluates to true, the dominant data dependence for this distance  $\Delta[4,-2]$  is set to 1 i.e.  $(-1,(2,1)) + (1,(3,-7)) \supseteq$

(1,(4,-2)) and the dependence (1,(4,-2)) is deleted from the dependence list S. Hence, the dependence list after step 3 will be

$$S = \{(-1,(1,-4)), (-1,(2,1)), (1,(3,-7))\}$$

Since dependencies (-1,(1,-4)) and (-1,(2,1)) have negative characteristic values these dependencies will not affect the scheduling of the code. In the third dependency (1,(3,-7)), because the inner loop index exceeds bounds of loop iterations it doesn't contribute anything to serialize the code. Thus the code is fully parallelizable and Figure 3.3 shows the scheduling for this example code.

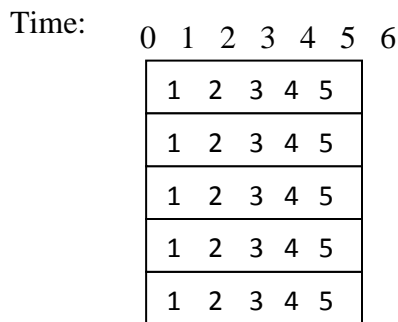


Figure 3.3: Scheduling of the example code.

### 3.9 Conclusion

We have discussed the notation that is used in representing dependencies in multidimensional loops and proved how the least inner loop coordinate subsumes all the other dependencies of the same outer loop distance. We have also developed an algorithm for reducing the number of dependencies in multidimensional loops to the minimum

possible number which improves the throughput of the system by eliminating unnecessary stalls in the pipeline. We have also explained how to eliminate redundant dependencies with an example by following the proposed algorithm. We can further minimize the dependencies in the program by applying loop transformation like loop skewing [4] to the code and applying our algorithm.

CHAPTER IV  
ELIMINATION OF DEPENDENCIES USING LOOP TRANSFORMATION  
METHODS

In this chapter we deal with elimination of dependencies in code by making use of both loop skewing transformation methods and the algorithm shown in Chapter 3. The basic idea is to learn the effect of loop skewing on our algorithm. First apply the loop transformation methods to the code. To the transformed code we apply the algorithm used in Chapter 3 and analyze the result produced. We analyze the effect of wave front transformations like horizontal skewing and vertical skewing (loop skewing) on the code.

#### 4.1 Vertical Shearing (Loop Skewing)

In vertical shearing we skew the inner loop with respect to the outer loop by some factor  $f$ . By doing this inner loop bodies execute in parallel since data dependencies are arranged horizontally. Vertical shearing can be represented in matrix notation

as  $\begin{bmatrix} 1 & 0 \\ f & 1 \end{bmatrix}$ [2]. The iteration space of Figure 2.3 after vertical shearing by a factor of one

is shown in Figure 4.1.

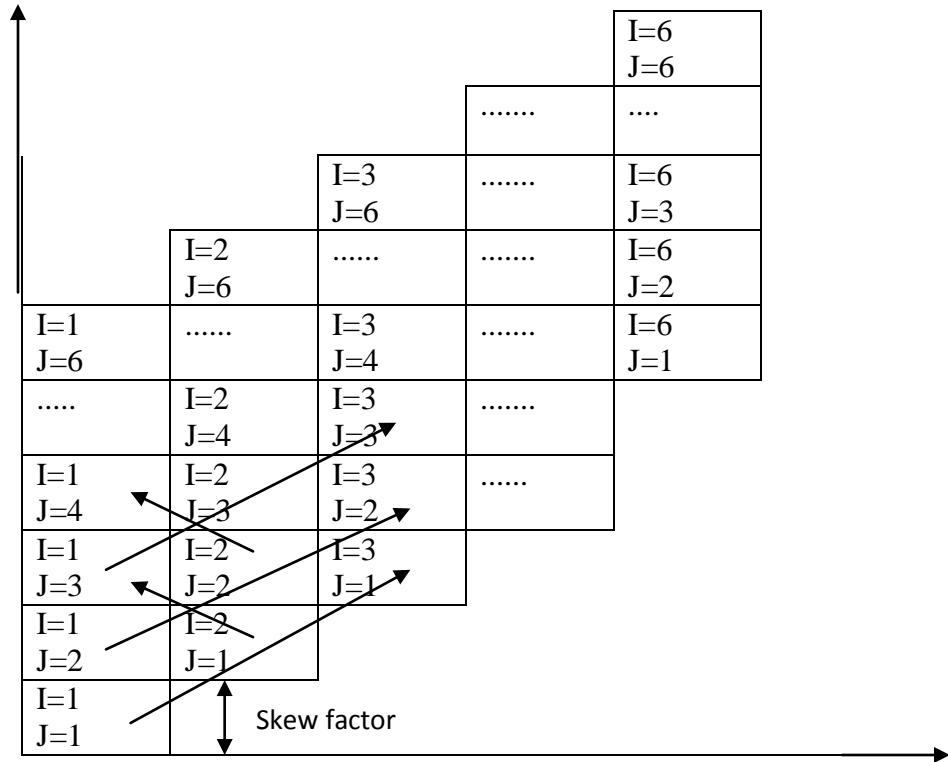


Figure 4.1: Iteration space after vertical shearing.

Theorem 4.1: Consider a multidimensional loop having a data dependence  $(\lambda, (d_x, d_y))$ , loop skewing the code with a factor  $f$  would change the coordinates of this dependence to  $(\lambda, (d_x, f \cdot d_x + d_y))$ .

Proof: Consider a two-dimensional loop with outer loop variable  $x$ , inner loop variable  $y$  and a data dependence  $(\lambda, (d_x, d_y))$ . Loop skewing  $y$  w.r.t.  $x$  with a factor of  $f$  can be shown in matrix notation as below.

$$\begin{bmatrix} 1 & 0 \\ f & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x + 0 \\ f \cdot x + y \end{bmatrix} = \begin{bmatrix} x \\ f \cdot x + y \end{bmatrix}$$

If we take new coordinates as  $i$  and  $j$ , that is  $\begin{bmatrix} x \\ f \cdot x + y \end{bmatrix} = \begin{bmatrix} i \\ j \end{bmatrix}$  then all occurrences of the variable  $x$  are replaced with  $i$  and  $j = f \cdot x + y$ . Therefore  $d_i = d_x$  and  $d_j = f \cdot d_x + d_y$  in the new notation. By replacing all of the loop bounds and array indices the old dependence  $(\lambda, (d_x, d_y))$  will be changed to  $(\lambda, (d_x, f \cdot d_x + d_y))$ . QED

## 4.2 Horizontal Shearing

In horizontal shearing[2] we skew the outer loop with respect to the inner loop by some factor  $f$ . This method is similar to loop skewing except that we force data dependencies to move forward along the original loop. Now the loop bodies which are arranged vertically can execute in parallel. Horizontal shearing can be represented in matrix notation as  $\begin{bmatrix} 1 & f \\ 0 & 1 \end{bmatrix}$ [2]. The iteration space of Figure 2.3 after horizontal shearing by a factor of one is shown in Figure 4.2.

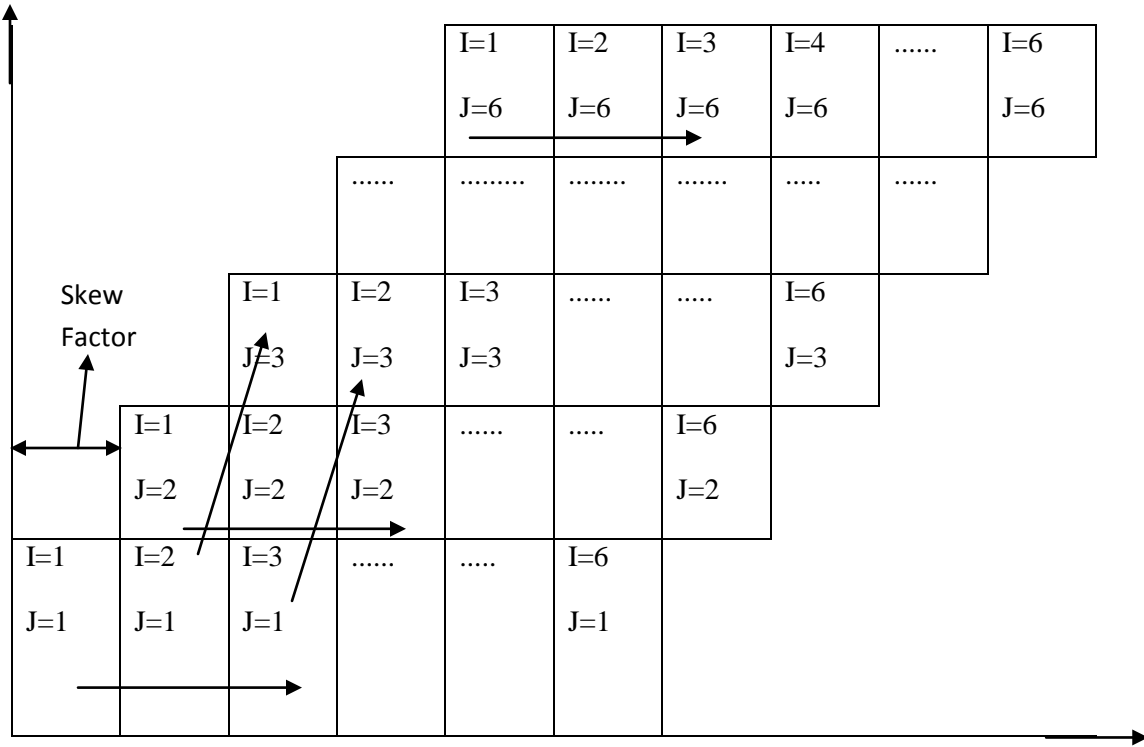


Figure 4.2: Iteration space after horizontal shearing.

Theorem 4.2: Consider a multidimensional loop having a data dependence  $(\lambda, (d_x, d_y))$ .

Horizontal skewing the code with a factor  $f$  would change the coordinates of this dependence to  $(\lambda, (d_x + f \cdot d_y, d_y))$ .

Proof: Consider a two-dimensional loop with outer loop variable  $x$ , inner loop variable  $y$  and a data dependence  $(\lambda, (d_x, d_y))$ . Horizontal skewing with a factor of  $f$  can be shown in matrix notation as below.

$$\begin{bmatrix} 1 & f \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x + f \cdot y \\ 0 + y \end{bmatrix} = \begin{bmatrix} x + f \cdot y \\ y \end{bmatrix}$$

If we take new coordinates as  $i$  and  $j$ , that is  $\begin{bmatrix} x + f \cdot y \\ y \end{bmatrix} = \begin{bmatrix} i \\ j \end{bmatrix}$ , then all

occurrences of the variable  $x$  are replaced with  $i - f \cdot y$  and occurrences of  $y$  are

replaced with  $j$ . Therefore  $d_i = d_x + f \cdot d_y$  and  $d_j = d_y$  in the new notation. By

replacing all of the loop bounds and array indices the old dependence  $(\lambda, (d_x, d_y))$  will be changed to  $(\lambda, (d_x + f \cdot d_y, d_y))$ . QED

### 4.3 Wavefronting

Wavefronting is another loop transformation technique which can transform a code having  $n$  loops with all the  $n$  loops carrying dependencies to code with at least  $n-1$  parallel loops. This is achieved by skewing the innermost loop with each of the other loops and moving the innermost loop to the outermost position. Wavefronting can be represented in matrix notation as  $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$ [4].

### 4.4 Combining Loop Skewing with Data Dependence Elimination

To analyze the effect of loop skewing on the dependence elimination algorithm we first find all loop carried dependencies in sample code after applying the dependence elimination algorithm. Next we skew the code with a factor  $f=1$ . Finally we change the indices of the array elements according to the new loop variables and find the loop carried dependencies in the code by again applying the dependence elimination algorithm to the skewed loop. The whole process is shown below with the help of sample code in Figure 4.3.

```

for i = 0 to n
{
  for j = 0 to n
  {
    a[i+1][j+1] = b[0][0]           /* Instruction 1 */
    a[i+2][j+3] = a[i][j]         /* Instruction 2 */
    a[i][j]      = a[i][j-2]       /* Instruction 3 */
    a[i-1][j-3]  = a[i+1][j+2] + a[i][j+2] /* Instruction 4 */
  }
}

```

Figure 4.3: Sample code with loop carried dependencies.

We have a total of 6 loop carried dependencies in the above code which are  $(-1,(1,1))$ ,  $(-2,(1,3))$ ,  $(-3,(1,-1))$ ,  $(-2,(1,1))$ ,  $(-1,(2,5))$ , and  $(-2,(2,1))$ . After applying the dependence elimination algorithm that we produced in Chapter 3 without loop skewing we will have only four data dependencies,  $(-1,(1,1))$ ,  $(-3(1,-1))$ ,  $(-1(2,5))$  and  $(-2,(2,1))$ . Now we skew the above code with a factor of one and find all loop carried dependencies. The loop variables in the above code are  $i$  and  $j$ . Loop skewing in matrix notation can be shown as

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} i \\ i + j \end{bmatrix} = \begin{bmatrix} I \\ J \end{bmatrix}$$

From above equation we get  $I = i$  and  $J = i + j$ . Therefore  $j = J - i$ . Now replace all occurrences of  $j$  with its equivalent i.e.  $J - i$ . The code in Figure 3.3 after changing the indexes is shown below in Figure 4.4.

```

for i = 0 to n
{
  for J = i to n + i
  {
    a[i+1][J-i+1] = b[0][0]           /* Instruction 1 */
    a[i+2][J-i+3] = a[i][J-i]       /* Instruction 2 */
    a[i][J-i]      = a[i][J-i-2]     /* Instruction 3 */
    a[i-1][J-i-3] = a[i+1][J-i+2] + a[i][J-i+2] /* Instruction 4 */
  }
}

```

Figure 4.4: Sample code with loop carried dependencies after loop skewing with skew factor,  $f = 1$ .

We now have 6 data dependencies which are  $(-1,(1,2))$ ,  $(-2,(1,4))$ ,  $(-3,(1,0))$ ,  $(-2,(1,2))$ ,  $(-1,(2,7))$ , and  $(-2,(2,3))$ . The next step in the process would be to apply the algorithm that we produced in Chapter 3 and analyze the output.

In step 1 of the algorithm we keep the dependencies with the maximum characteristic value for a distance and delete all others. Thus we are left with five dependencies after the first step which are  $(-1,(1,2))$ ,  $(-2,(1,4))$ ,  $(-3,(1,0))$ ,  $(-1,(2,7))$ , and  $(-2,(2,3))$ . In fact these are the transformed versions of the same dependencies derived above, prior to skewing.

In step 2 of the algorithm we keep the dependencies with largest characteristic values and minimum inner loop iteration index for each outer loop and delete all the remaining dependencies. Thus the dependence list after step 2 will be  $(-1,(1,2))$ ,  $(-3,(1,0))$ ,  $(-1,(2,7))$ , and  $(-2,(2,3))$ . Again, these are the transformed versions of  $(-1,(1,1))$ ,  $(-3,(1,-1))$ ,  $(-1,(2,5))$ , and  $(-2,(2,1))$ , the dependencies that emerged when we applied our method without skewing.

The dominant dependence does not eliminate either of the remaining dependencies. Thus the total number of dependencies after skewing the code and applying our algorithm is same as without applying loop skewing in this example. We formalize this idea below.

Lemma 4.1: The maximum characteristic value for a given distance after loop skewing corresponds to the maximum characteristic value before loop skewing.

Proof: Consider data dependencies  $(\lambda_1, (d_{x1}, d_{y1}))$ ,  $(\lambda_2, (d_{x2}, d_{y2}))$ ,  $(\lambda_3, (d_{x2}, d_{y2}))$ , and  $(\lambda_4, (d_{x4}, d_{y4}))$  where  $\lambda_3 > \lambda_2$ . By applying elimination algorithm, dependence  $(\lambda_2, (d_{x2}, d_{y2}))$  get eliminated (since  $\lambda_3 > \lambda_2$ ). So maximum characteristic value for distance  $(d_{x2}, d_{y2})$  is  $\lambda_3$ . By loop skewing with a factor  $f$  these dependencies would move to  $(\lambda_1, (d_{x1}, f \cdot d_{x1} + d_{y1}))$ ,  $(\lambda_2, (d_{x2}, f \cdot d_{x2} + d_{y2}))$ ,  $(\lambda_3, (d_{x2}, f \cdot d_{x2} + d_{y2}))$ , and  $(\lambda_4, (d_{x4}, f \cdot d_{x4} + d_{y4}))$ . Even now the maximum characteristic value for distance  $(d_{x2}, f \cdot d_{x2} + d_{y2})$  is  $\lambda_2$  because loop skewing moves all the dependencies for a given distance to same position which adds a constant value to all the inner loop iteration index.

Lemma 4.2: The dependence with minimum inner loop iteration index after loop skewing corresponds to the dependence with minimum inner loop index before skewing.

Proof: Consider data dependencies  $(\lambda_1, (d_{x1}, d_{y1}))$ ,  $(\lambda_2, (d_{x1}, d_{y2}))$ ,  $(\lambda_3, (d_{x2}, d_{y3}))$ , and  $(\lambda_4, (d_{x2}, d_{y4}))$  where  $d_{y2} < d_{y1}$  and  $d_{y4} < d_{y3}$ . By applying elimination algorithm, dependence  $(\lambda_1, (d_{x1}, d_{y1}))$  and  $(\lambda_3, (d_{x2}, d_{y3}))$  get deleted since we keep dependencies with only minimum inner loop iteration index for dependencies with same outer loop index. After loop skewing also we see that transformed versions of dependencies  $(\lambda_1, (d_{x1}, d_{y1}))$  and

$(\lambda_3, (d_{x2}, d_{y3}))$  get deleted because loop skewing adds a constant value to all the inner loop iteration index's so the relative difference between the distances remain same after and before loop skewing.

Lemma 4.3: The dominant data dependence for a given distance before loop skewing corresponds to the dominant data dependence after loop skewing.

Proof: In dominant data dependence, for two dependencies to subsume another dependence it needs to satisfy two requirements. First, the sum of outer loop indexes of two dependencies should be one greater than subsumed dependency. Second, the sum of the inner loop indexes of two dependencies should be less than or equal to the subsumed dependence.

After loop skewing we see that first requirement is obviously solved because loop skewing doesn't change the outer loop iteration index. The second requirement is also solved because the sum of inner loop iteration index of decomposed distances is always less than or equal to the subsumed dependency since loop skewing adds a constant value to inner loop index of subsumed dependency and other dependencies.

Theorem 4.3: Loop skewing has no effect on the outcome of our dependence elimination algorithm.

Proof: A dependence can be subsumed by our algorithm in one of three ways: by the dependence having maximum characteristic value for its distance; by the dependence with the same outer loop iteration index but the minimum inner loop iteration index; and by the sum of dominant dependencies whose distance adds to the distance of the given

dependence. By the Lemmas none of these are changed by loop skewing. Thus, if a dependence is subsumed before skewing, its transformed equivalent is subsumed after skewing. We thus say that loop skewing has no effect on the outcome of the algorithm. QED

#### 4.5 Combining Horizontal Skewing with Data Dependence Elimination

In this section we deal the effect of horizontal skewing on the elimination algorithm. We follow the same steps that we did in Section 4.3. We show the whole process by considering the sample code in Figure 3.3. We have 6 data dependencies in the code which get reduced to 2 after applying the dependence elimination algorithm. Now we will see the effect of horizontal skewing with skew factor one on the algorithm. The matrix notation of the horizontal skewing is shown below.

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} i + j \\ j \end{bmatrix} = \begin{bmatrix} I \\ J \end{bmatrix}$$

From the above equation we get  $I = i + j$  and  $J = j$ . Therefore  $i = I - j$ . By replacing all occurrences of  $i$  with its equivalent (i.e.  $I - j$ ) and loop interchanging we have the modified code shown below in Figure 4.5.

```

for j = 0 to n
{
  for I = j to n + j
  {
    a[I-j+1][j+1] = b[0][0]           /*Instruction1 */
    a[I-j+2][j+3] = a[I-j][j]       /*Instruction 2 */
    a[I-j][j]      = a[I-j][j-2]     /*Instruction 3 */
    a[I-j-1][j-3] = a[I-j+1][j+2] + a[I-j][j+2] /* Instruction 4*/
  }
}

```

Figure 4.5: Sample code with loop carried dependencies after horizontal skewing with skew factor  $f = 1$ .

We have a total of 5 data dependencies in the above code which are  $(-1,(1,2))$ ,  $(-2,(3,4))$ ,  $(-2,(1,2))$ ,  $(-1,(5,7))$ , and  $(-2,(1,3))$ . As we did for loop skewing, we now apply the dependence elimination algorithm to the above code.

In step 1 of the algorithm we keep the dependencies with the maximum characteristic value for a given distance and delete all others. Thus we are left with five dependencies after the first step which are  $(-1,(1,2))$ ,  $(-2,(3,4))$ ,  $(-1,(5,7))$ , and  $(-2,(1,3))$ .

In step 2 of the algorithm we keep the dependencies with largest characteristic values and minimum inner loop iteration index for each outer loop and delete all remaining dependencies. Thus the dependence list after step 2 will be  $(-2,(3,4))$ ,  $(-1,(1,2))$ , and  $(-1,(5,7))$ .

The dominant dependence does not eliminate any of the remaining dependencies. Thus the total number of dependencies after horizontal skewing the code and applying our algorithm is 3 which is less than before applying horizontal skewing.

## 4.6 Combining Wavefronting with Data Dependence Elimination

In this section we deal with the effect of wavefronting on the elimination algorithm. We follow the process that we did for the other two skewing techniques. We explain the process using the sample code in Figure 4.3 which has 6 data dependencies that get reduced to 2 after applying the dependence elimination algorithm. Now we will see the effect of wavefronting on the algorithm. The matrix notation of wavefronting is shown below.

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

From the above equation we get  $i = J$  and  $I = i + j$  which implies  $j = I - i = I - J$  ( since  $i = J$  ). Now by replacing all occurrences of  $i$  with its equivalent (i.e.  $I - j$ ),  $j$  with its equivalent ( i.e.  $i$  ) and by loop interchanging we have the modified code shown in Fig. 4.6.

```

for I = 2 to 2n
{
  for J = Max(1 , n - I) to Min(n , |I - 1|)
  {
    a[J+1][I-J+1] = b[0][0]           /* Instruction 1 */
    a[J+2][I-J+3] = a[i][j]           /* Instruction 2 */
    a[J][I-J]      = a[J][I-J-2]       /* Instruction 3 */
    a[J-1][I-J-3]  = a[J+1][I-J+2] + a[J][I-J+2] /* Instruction 4 */
  }
}

```

Figure 4.6: Sample code with loop carried dependencies after wavefronting.

We have a total of 6 data dependencies in the above code which are  $(-1,(2,1))$ ,  $(-2,(4,1))$ ,  $(-3,(0,1))$ ,  $(-2,(2,1))$ ,  $(-1,(7,2))$ ,  $(-2,(3,2))$ . As we did for loop skewing, we now apply the dependence elimination algorithm to the above code.

In step 1 of the algorithm we keep the dependencies with the maximum characteristic value for a given distance and delete all others. Thus we are left with five dependencies after the first step which are  $(-1,(2,1))$ ,  $(-2,(4,1))$ ,  $(-3,(0,1))$ ,  $(-1,(7,2))$ , and  $(-2,(3,2))$ .

In step 2 of the algorithm we keep the dependencies with largest characteristic values and minimum inner loop iteration index for each outer loop value and delete all the remaining dependencies. The dependence list after step 2 will be same as after step 1 since none of the dependence get eliminated.

In step 3 of the algorithm we keep only the dominant dependencies in the code and eliminate all that remain. In our dependence list none of the dependency get eliminated by applying dominant dependencies. Thus the total number of dependencies remaining after applying the elimination algorithm to the wave fronting code is many more than without applying wavefronting.

#### 4.7 Conclusion

In this chapter we discussed the effect of loop transformation techniques like loop skewing(vertical skewing), horizontal skewing on the elimination algorithm. From our experiments we can conclude that loop skewing has no major effect on elimination algorithm since we get the same dependencies with and without applying loop skewing.

This has been proved in Theorem 4.3. We also inferred that horizontal skewing and wavefronting have no predictable effect on the elimination algorithm. So we can only conclude that further study is needed to know the effect of these transformations on elimination algorithm. We have also developed proofs in Theorems 4.1 , 4.2 and 4.3 that loop skewing move the dependencies to new positions in a predictable pattern which is based on the skew factor.

## CHAPTER V

### CONCLUSIONS AND FUTURE WORK

In this chapter, we review the contributions from this research and provide direction for future improvements that can be made to improve the performance of our work.

#### 5.1 Contributions

In this research we mainly focused on representation and minimization of data dependencies in multi-dimensional loops. Through our research we have demonstrated that only those dependencies which remain after applying the elimination algorithm need to be considered while scheduling the code. Here we outline all the contributions of our work.

We have gone through many ways of representing multi-dimensional data dependencies and finally came up with one particular representation which holds all the details of dependence. Next, we developed a formula with supporting proof for sums of data dependencies in multi-dimensional loops. We showed how two different data dependencies can eliminate another dependence.

We also discussed how to handle negative distances in the dependence representation and proved how the dependence with the least negative distance in the innermost loop subsumes all the other dependencies of the same outer loop distance. Finally, we developed a three step algorithm for minimizing data dependencies in multi-dimensional loops using all the notations and proofs we discussed above.

In Chapter 4, we took one step forward and tried to further minimize dependencies to improve the efficiency of the elimination algorithm by combining it with loop transformation techniques. The transformation techniques we used in our research are loop skewing, horizontal skewing, and wavefronting. After thorough study on the effect of transformations on elimination algorithm, we concluded that loop skewing basically doesn't have any effect on the algorithm because in all the cases the dependencies remain the same before and after applying loop skewing.

For the other two transformations there is some effect on the elimination algorithm but not the effect that we are exactly looking for because dependencies get increased instead of reducing after applying these transformations. Since we do not have proper evidence to comment further we conclude that further study is required to know the effect of these transformations on elimination algorithm.

## 5.2 Future Work

In our research we selected loop skewing, horizontal skewing, and wavefronting as transformation techniques to be combined with the elimination algorithm and concluded that none of the three has showed the desired effect. Future work could include

developing a better way to deal with the inner loops, studying the effect of other transformations like loop tiling, loop fusion, and loop peeling on the elimination algorithm and further work can be made on studying of these techniques with non-unit array index coefficients and trapezoidal rather than rectangular execution spaces.

In our research we have studied the effect of combining transformation techniques with the elimination algorithm by applying the transformation to the loop first and then applying the elimination algorithm. Since we found that horizontal skewing and wavefronting have some effect with this method future work can include how to apply wavefronting and horizontal skewing to the elimination algorithm effectively so that the application of these techniques reduce the number of data dependencies in the code.

## REFERENCES

- [1] Wolfe, M., Loop Skewing: The Wavefront Method Revisited, International Journal of Parallel Programming, Springer Netherlands, pp.279-293,(1986).
- [2] ] Kyoko Iwasawa, Alan Mycroft. Choosing Method of the Most Effective Nested Loop Shearing for Parallelism. In Proceeding of the Eighth International Conference on Parallel and Distributed Computing, Applications and Technologies, pp.267-276 ,(2007).
- [3] Chun Xue, Zili Shao, Edwin H.-M. Sha. Maximize Parallelism Minimize overhead for Nested Loops via Loop Striping. The journal of VLSI signal processing-systems for signal,image, and video technology, Vol:47, pp.153-167,(2007).
- [4] Micheal E.Wolf, Monica S.Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. IEEE Transactions on Parallel and Distributed Systems, Volume 2: pp.452-471 ,(1991).
- [5] Sam Jin Jeong., Maximizing Parallelism for Nested Loops with Non-uniform Dependencies. Computational Science and Its Applications – ICCSA. Volume 3046: pp. 213 – 222, (2004).
- [6] Michael E.Wolf, Dror E. Maydan, and Ding-Kai Chen. Combining Loop Transformations Considering Caches and Scheduling. International Journal of Parallel Programming, Volume:26 pp. 479-503,(1998).
- [7] Eduard Ayguade, Jordi Torres. Partitioning the statement per iteration space using non-singular matrices. In Proceedings of the 7<sup>th</sup> international conference on Supercomputing, pages:407-415, (1993)
- [8] Jordi Torres, Eduard Ayguade, Jesus Labarta, Mateo Valero. Revisiting Framework of Linear Loop Transformation. In Proceedings of Fourth Euromicro Workshop on Parallel and Distributed Processing, IEEE Computer Society. (1995).

- [9] Banerjee U., Loop Transformations for Restructuring Compilers- Loop Parallelization. Klumer Academic Publishers, Norwell, Massachusetts. (1994).
- [10] Kathryn S. McKinley, Steve Carr, Chau-Wen Tseng. Improving data locality with loop transformations. ACM Transactions on Programming Languages and Systems, Volume:18 pp 424-453, (1996).
- [11] T. Jacobson and G. Stubbendieck. Dependency analysis of for-loop structures for automatic parallelization of c code. Mathematics and Computer Science Department South Dakota School of Mines and Technology, pp. 1-13 , (2002).
- [12] Timothy W. O'Neil, Edwin H.M Sha. Minimizing inter-iteration dependencies for loop pipelining. In Proceeding of ISCA 13th International Conference On Parallel and Distributed Computing System, pp 412-417, (2000).
- [13] Krishna Chaitanya Chakilam, Sukumar Reddy Anapalli, Timothy W. O'Neil. Minimizing Inter-Iteration Dependencies in Multi-Dimensional Loops. In Proceedings of ISCA 22<sup>nd</sup> International Conference On Parallel and Distributed Computing and Communication Systems. (2009).
- [14] Ken Kennedy, John R.Allen, Optimizing compilers for modern architectures: a dependence-based approach, Morgan Kaufmann Publishers Inc., San Francisco, CA,2001.