

# Minimizing Inter-Iteration Dependencies in Multi-Dimensional Loops

Krishna Chaitanya Chakilam  
Dept. of Computer Science  
The University Of Akron  
Akron, Oh, 44325, USA  
kc51@uakron.edu

Sukumar Reddy Anapalli  
Dept. of Computer Science  
The University Of Akron  
Akron, Oh, 44325, USA  
sa46@uakron.edu

Timothy W. O'Neil  
Dept. of Computer Science  
The University Of Akron  
Akron, Oh, 44325, USA  
toneil@uakron.edu

## Abstract

Since compiler spend most of the time in executing loop nests because of the dependences, minimizing dependences 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 dependences by combining other dependencies and method two eliminates dependences by making use of the distance of the dependence. To conclude we provide an algorithm for minimizing the number of dependences in multi-dimensional loops.

*Index Terms:* multi-dimensional dependencies, loop carried dependences.

## 1 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 techniques available to optimize a compiler to get through with these problems.

Until today there has been a lot of research done on how to minimize loop-carried dependencies in multidimensional loops. Most of the works introduces different transformation techniques such as loop unrolling, loop splitting, loop fusion, loop skewing, wavefronting and many others.

A loop parallelizing method using horizontal and vertical shearing is described in [3]. The article tries to achieve the advantages of horizontal shearing, which is still an unexplored area. 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 on finding where the code consist the chunk 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 [4] 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, loop interchange is explained in [5]. The authors claim that, existing compilers must choose appropriate transformation 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 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 this paper we will 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 [1]. The rest of the paper is organized as follows. Chapter 2 discusses the basic concepts of data dependences, loop transformation, etc. which lay foundation for this thesis. It also explains new terminology such as subsumption, characteristic value, sums of dependences, and dominant

data dependence which are used in developing redundant dependence elimination algorithm. Chapter 3 presents a new representation for dependences in multidimensional loops, generates proof for sums of dependences in two dimensional loops and provides an algorithm for minimizing dependences. In Chapter 4, we draw the conclusion from this thesis and provide recommendations for future work.

## 2 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 three different types of dependencies: Read-After-Write (RAW), Write-After-Read (WAR), and Write-After-Write (WAW). The WAR and WAW hazards arise only because two instructions try to make use of same memory location at same time. If the register names are changed so that conflict doesn't happen these hazards can be eliminated.

We are concerned only with RAW hazards, the most common kind of hazard. It exists when instruction  $j$  tries to read a location before instruction  $i$  writes to it. In other words if we assume instruction  $i$  executes at time  $t_1$  and instruction  $j$  executes at time  $t_2$  then 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*. 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 register. Therefore, we mainly concentrate on eliminating these kinds of dependencies.

### 2.1 Intra and Inter Iteration dependencies

If the dependence exist between two instructions of the same iteration then it is called an *intra-iteration* dependence. If the dependence exist 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 in an intra-iteration dependence is zero.

Consider the following example code in Fig 2.1 and we will see more clearly the kinds of dependencies that exist in the code. By unrolling the loop we can see there are 8 RAW hazards in the code. They are  $(1 \rightarrow 2, 0)$ ,  $(1 \rightarrow 4, 0)$ ,  $(2 \rightarrow 3, 1)$ ,  $(2 \rightarrow 5, 1)$ ,  $(3 \rightarrow 2, 1)$ ,  $(3 \rightarrow 4, 1)$ ,  $(4 \rightarrow 3, 2)$  and  $(4 \rightarrow 5, 2)$ . The first part in this notation shows the instructions having the dependence and the second part shows the inter-iteration distance of the instructions (dependence distance). We eliminate some of the dependencies by applying the elimination algorithm from [1].

```

for i = 1 to 5
{
  a[i] = x+1      /* Instruction 1*/
  b[i+1] = a[i]+1; /* Instruction 2*/
  a[i+1] = b[i];  /* Instruction 3*/
  b[i+2] = a[i]-1; /* Instruction 4*/
  x = b[i];      /* Instruction 5*/
}

```

Fig 2.1: Example code with loop-carried dependencies.

### 2.2 Characteristic Value and Subsumption:

Not all dependencies should be considered to increase the parallelism in the code. Some techniques like subsumption and characteristic data dependence, which eliminate data dependencies in code, are discussed in depth in [1]. By implementing these concepts in the elimination algorithm from [1] we can greatly reduce the number of data dependencies.

#### 2.3 Characteristic value

Let us consider a data dependence  $(X \rightarrow Y, d)$ . If  $X$  executes at time  $T_x$  and  $Y$  executes at time  $T_y$  then the characteristic value of the data dependence is defined as difference in start times of the two instructions, i.e.  $T_x - T_y$ . For example, in the dependence  $(2 \rightarrow 4, 1)$ , the characteristic value is -2.

#### 2.4 Subsumption

Consider a dependence from  $a$  to  $b$ , where  $a$  executes at time  $T_i$  and  $b$  executes at time  $T_j$ . We can say this dependence is fulfilled if  $T_i < T_j$  or  $b$  executes before  $a$ . Whenever fulfillment of dependence  $A$  automatically satisfies dependence  $B$  then we say that  $A$  *subsumes*  $B$  and is denoted as  $A \supseteq B$ . For example, in Fig. 2.1 we can see that  $(2 \rightarrow 3, 1) \supseteq (2 \rightarrow 5, 1)$ .

### 3 Multi-dimensional dependencies

This chapter includes the new type of notation for the dependencies in multi-dimensional loops, basic terminology which is specific for multidimensional loops and with supporting proofs we discuss the techniques for eliminating redundant dependencies. The proposed algorithms are explained with the help of an example.

#### 3.1 Dependence notation in multi-dimensional loops

Unlike the one-dimensional case, the notation for dependencies in multi-dimensional loops cannot be represented in the form of  $(\lambda, d)$  because the distance in a multi-dimensional 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 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];
  }

```

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

#### 3.2 Negative Co-ordinates

The co-ordinates 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 before the start of the second iteration of the outer loop. There exists no dependence when the outer loop co-ordinate is negative. The dependence with the least negative inner loop coordinate subsumes all other dependencies of the same outer loop coordinate. For example, the dependence  $(1, (2, -2))$  subsumes dependencies  $(2, (2, -1))$  and  $(1, (2, 1))$ . This is formally proven as Lemma 3.1 of [6].

Lemma 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  $B \supseteq A$ . ■

Thus the least inner loop distance and largest characteristic value subsumes all dependencies of the same outer loop distance.

#### 3.3 Sums of Data Dependencies in Multiple dimensional loops

So far the dependencies in two-dimensional loops are represented as  $(\lambda_a, (d_x, d_y))$ , which is equivalent to the one-dimensional representation as  $(\lambda_a, (d_{ax}-1)*n + d_{ay})$  where 'n' represents the total number of times the outer loop executes. Here the distance across the loops is represented as a single value instead of a co-ordinate pair. Following this notation we prove the following as Theorem 3.1 of [6].

Theorem: The sum of two 2-dimensional data dependencies is

$$\begin{aligned}
 & (\lambda_a, (d_{ax}, d_{ay})) + (\lambda_b, (d_{bx}, d_{by})) \\
 &= (\lambda_a + \lambda_b + 1, (d_{ax} + d_{bx} - 1, d_{ay} + d_{by})). \quad \blacksquare
 \end{aligned}$$

The sum can be inductively shown to be

$$\sum_{i=1}^n Ai = ((\sum_{i=1}^n \lambda_i + 1) - 1, (\sum_{i=1}^n dx_i - 1) + 1, \sum_{i=1}^n dy_i)$$

#### 3.4 Dominant Data Dependencies in multi-dimensional loops

The sums of data dependencies discussed in the last section demonstrate how an individual dependence can subsume all other dependencies of the same distance which is denoted as  $\Lambda$ . 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 an outer loop coordinate and 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.

### 3.5 Algorithm for Eliminating Redundant Data Dependencies in multidimensional loops

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

This module (Fig 3.2) finds the maximum characteristic value for each distance and deletes all the dependencies which have characteristic value less than the maximum.

```

for i = 1 to M do
  for j = a to b do
     $\Lambda[i,j] \leftarrow -\infty$ 
    for all data dependencies in  $(\lambda, (d_i, d_j))$ 
      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
end for

```

**Fig 3.2. Algorithm for finding maximum characteristic value.**

This module (Fig 3.3) 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 1 in section 3.2.

```

for i = 1 to D
  for j = a to b
     $n \leftarrow \infty$ 
     $v \leftarrow -\infty$ 
    for all dependencies with outer loop index i
      if  $d_y < n$  then
        if  $\lambda > v$  then
          if  $n < \infty$  and  $v > -\infty$  then
            delete data dependence  $(v, (i, n))$ 
          end if
           $n \leftarrow d_y$ 
           $v \leftarrow \lambda$ 
        end if
      else
        delete data dependence  $(\lambda, (i, d_y))$ 
      end if
    end for
  end for
end for
end for

```

**Fig 3.3. Algorithm for eliminating dependencies in same outer loop distance.**

In this module (Fig 3.4) we find the largest sum of characteristic value whose outer loop distances add to  $d_x$  and inner loop distances add to  $d_y$ . A dependence is eliminated if this sum is greater than the characteristic value of the dependence and 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  $(\lambda, (d_i, d_j))$  in variable  $P_i$ , i.e. if the first coordinates of both the dependencies sums up to the first coordinate of the dependency  $(\lambda, (d_i, d_j))$ . 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  $(\lambda, (d_i, d_j))$ . If this happens we delete the

dependency  $(\lambda, (d_i, d_y))$  and the dominant data dependency for this distance  $(d_i, d_y)$  is the sum of the characteristic values of the dependencies in  $P_i$ .

In this module (Fig 3.4) we find the largest sum of characteristic value whose outer loop distances add to  $d_x$  and inner loop distances add to  $d_y$ . A dependence is eliminated if this sum is greater than the characteristic value of the dependence and 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 sums up to the first coordinate of the dependency  $(\lambda, (d_i, d_y))$ . In the second part (Fig 3.5) 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 the dominant data dependency for this distance  $(d_i, d_y)$  is the sum of the characteristic values of the dependencies in  $P_i$ .

```

 $\Delta[1, a] = \Lambda[1, a]$ 
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 =  $\Delta[j] + \Delta[i-j] + 1$ 
         $P_{\lambda i} = (((\lambda_1, (d_j, d_{y1})) + (\lambda_2, (d_{i-j}, d_{y2}))))$ 
         $P_{x_i} = d_j + d_{i-j}$ 
         $P_{y_i} = d_{y1} + d_{y2}$ 
      end if
    end if
  end for
end for

```

**Fig 3.4 Algorithm for finding Dominant Dependencies (step 1)**

```

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

```

**Fig 3.5 Algorithm for finding Dominant Dependencies (step 2)**

### 3.6 Example to demonstrate the algorithm

The algorithm is demonstrated for the sample program below.

```

do i = 1 to 5
  do j = 1 to 5
    a[i+1][j+1] = a[1][1];
    b[i+4][j+2] = a[i][j+5] + a[i-1][j]
    a[i+3][j-2] = a[i+3][j-3]
    a[i-3][j-3] = a[i][j+1] + b[i+3][j]
    c[i+1][j+1] = a[i][j+1]
    c[i][j] = b[i+1][j+3] + b[i][j+4]
  end do
end do

```

**Fig 3.6: Sample code representing loop carried dependencies in two-dimensional loop.**

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

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

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 dependencies  $(-4, (1, 0))$  and  $(-2, (3, -3))$  are deleted since the characteristic values of these dependencies are less

than  $(-3,(1,0))$  and  $(-1,(3,-3))$  respectively. Thus, the dependency list  $S$  at the end of step 1 will be

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

In step 2 the dependence with largest characteristic value and smallest  $J$  loop distance for each  $I$  value is stored and the remaining dependencies 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  $P_{xi}$  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 and the dependence  $(1,(4,-2))$  is deleted from the dependence list  $S$ . Hence, the dependence list after step 3 will be

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

#### 4 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 transformations like loop skewing [2] to the code and applying our algorithm.

#### References

- [1] Timothy W. O'Neil and Edwin H.M Sha. "Minimizing inter-iteration dependencies for loop pipelining." Proc. ISCA 13th International Conference on Parallel and Distributed Computing Systems, pp 412-417, 2000.
- [2] M. Wolfe, "Loop Skewing: The Wavefront Method Revisited," International Journal of Parallel Programming, Springer, Netherlands, pp. 279-293, 1986.
- [3] Kyoko Iwasawa and Alan Mycroft. "Choosing Method of the Most Effective Nested Loop Shearing for Parallelism." Proc. Eighth International Conference on Parallel and Distributed Computing, Applications and Technologies, pp.267-276, 2007.
- [4] Chun Xue, Zili Shao and Edwin H.-M. Sha. "Maximize Parallelism Minimize overhead for Nested Loops via Loop Striping." Journal of VLSI signal processing systems for signal, image, and video technology, Vol. 47, pp.153-167, 2007.
- [5] M.E.Wolf and M.S.Lam. "A Loop Transformation Theory and an Algorithm to Maximize Parallelism." IEEE Transactions on Parallel and Distributed Systems, Vol. 2, pp.452-471, 1991.
- [6] Krishna Chaitanya Chaklam. Representing and Minimizing Multidimensional Dependencies. M.S.C.S. Thesis, Dept. of Computer Science, The University of Akron, 2009.