Optimizing Cycle Shrinking Algorithm

B. Tech Project Report

Sameep Bagadia
Roll no: 100050003

under the guidance of Prof. Supratim Biswas

Department of Computer Science and Engineering
Indian Institute of Technology Bombay

Abstract

In this report we present an optimal algorithm for cycle shrinking transformation for parallelizing compilers. We first give a motivation for a better algorithm by pointing out that the extended cycle shrinking algorithm is not optimal when there is no dominating dependence distance. Then we give our algorithm to generate the partitions in the iteration space of loops in constant dependence distance case. We prove the correctness and optimality of our approach. We also give a method to calculate the number of partitions that our algorithm will generate without actually creating the partitions. We then proceed to give an algorithm to generate loops from the partitions we created. We also prove the correctness of this algorithm. Then we consider the variable dependence distance case and give a better algorithm than extended cycle shrinking algorithm. We also prove that the number of partitions generated by our algorithm is optimal. We also give some ideas for parallelizing our algorithm.

Contents

Chapter 1

Introduction

A parallelizing compiler converts a sequential program to a semantically equivalent parallel program. For this transformation to be correct, data dependences must be satisfied. Therefore, data dependence analysis is done. There are various tests for data dependence like GCD test, Banerjee test, Delta test etc. [1]

A data dependence graph (DDG) is a directed graph with each statement as a vertex and an edge from statement S_i to S_j if there is a dependence $S_i \Delta S_j$.

Based on the DDG, transformations are applied to the sequential code. There are many transformations like loop fusion, loop skewing, loop interchange etc. [1]. But most of these transformations work only when the data dependences do not form a cycle. In the case when there is a cyclic dependence straightforward parallelization cannot be done. In such cases, cycle shrinking algorithm can be used.

Chapter 2

Literature Survey

2.1 Cycle Shrinking Algorithm

Cycle shrinking algorithm [2] allows loops with cyclic data dependencies to be partially parallelized when the dependence distances are known. Minimum distance between source and sink for a dependence is known as dependence distance. This is a vector for nested loops.

The idea of cycle shrinking has been extended in three ways:

1. **Simple shrinking**: Minimum dependence distance for each nest level is calculated separately and then partitions are created based in these values.
2. **Selective shrinking**: Outermost level with positive dependence distance is selected and then simple shrinking is applied only at this level. All lower level loops are done in parallel.
3. **True distance shrinking**: The partition is made based on the actual number of iterations between the source and the sink ("true distance") of the dependencies considering the loop bounds at each nest level.

2.1.1 Example

Consider the following example:

```
DO I = 0 to 9
    DO J = 0 to 9
        A[I+3, J+4] = B[I, J]
        B[I+2, J+3] = A[I, J]
    ENDO
ENDO
```

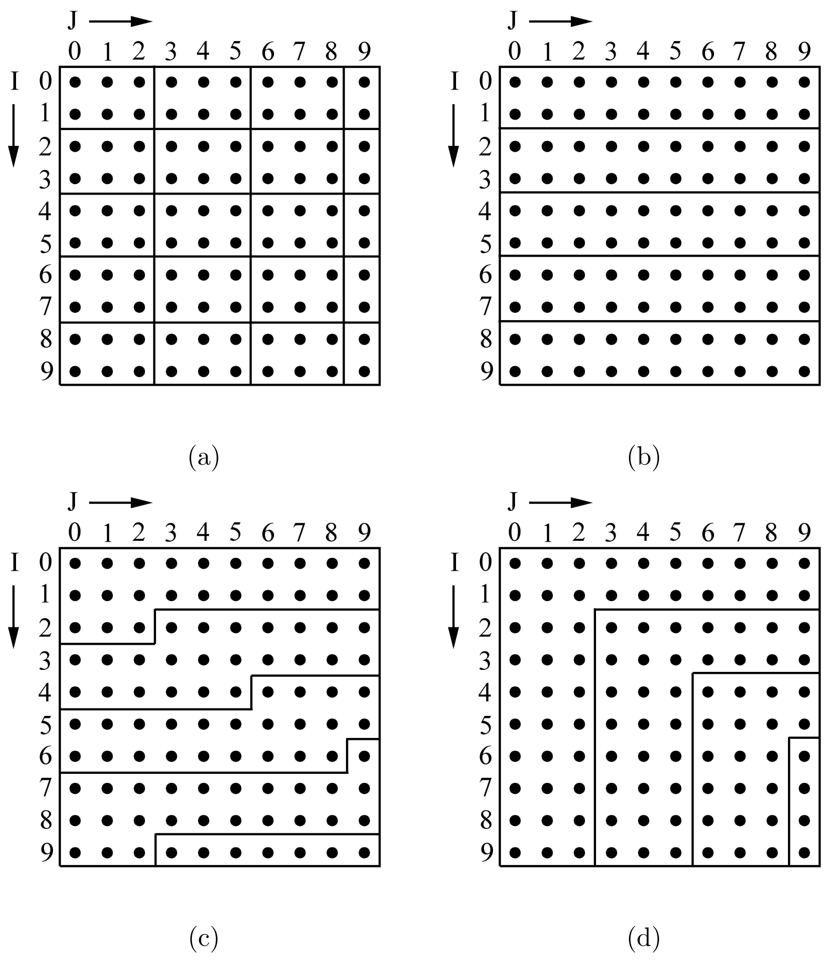


Figure 2.1: (a) Partitions by simple shrinking. (b) Partitions by selective shrinking. (c) Partitions by true distance shrinking. (d) Partitions by extended cycle shrinking algorithm

The two dependence distances are (3, 4) and (2, 3). The partitions created by cycle shrinking algorithm are shown in the figure 2.1.

However, a greater degree of parallelism can be achieved if the partitions are made as shown in the figure 2.1d. It reduces the number of partitions. Extended cycle shrinking algorithm creates paritions in this manner.

2.2 Extended Cycle Shrinking Algorithm

Extended Cycle Shrinking Algorithm [3] has two cases, when there are only constant dependence distances and when there are variable dependence distances.

2.2.1 Constant Dependence Distances

If two array accesses have a dependence among them then they are called as a reference pair. Consider the case where all the reference pairs have only constant dependence distances. The dependence distance vector of a reference pair R is denoted as $\Phi(R)$ and k^{th} component of the dependence distance $\Phi(R)$ is denoted as $\Phi_k(R)$. Then we define the dependence distance of vector of loop L denoted by $\Phi(L)$ as given below:

For each nest k of the loop,

- $\Phi_k(L) = 0$ if there exists some reference pair R for which $\Phi_k(R) = 0$, or there exist two reference pairs R and R' such that $\Phi_k(R)$ and $\Phi_k(R')$ have opposite signs.
- $\Phi_k(L) = min(|\Phi_k(R)|)$ if sign($\Phi_k(R)$) is positive for all R, and
- $\Phi_k(L) = -min(|\Phi_k(R)|)$ otherwise.

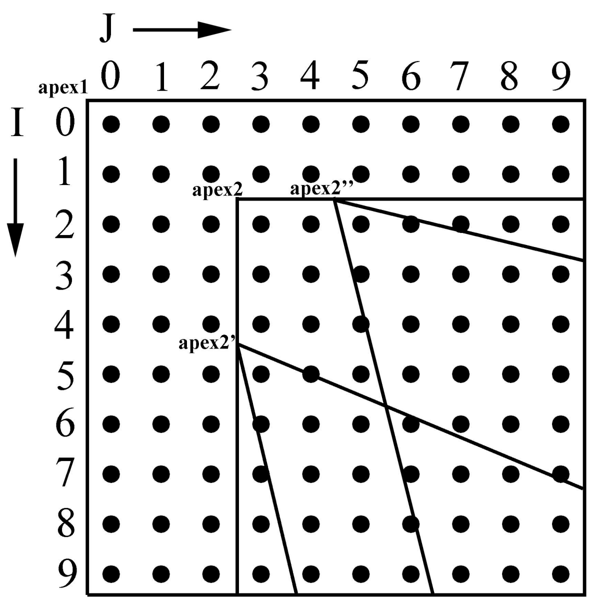
The partitions are created using $\Phi(L)$. The k^{th} coordinate of the i^{th} apex point of the partition is $(i-1) * \Phi_k(L)$ if $\Phi_k(L) > 0$, otherwise the apex point for that index is calculated from the other end of loop bound and the direction of partition is also opposite.

For example of figure 2.1d,
$\Phi(L) = (2, 3)$
Therefore the apex points are (0, 0), (2, 3), (4, 6) and (6, 9).

2.2.2 Variable Dependence Distances

In the case of variable dependence distances, only those cases are considered where each source can have only one sink for a given reference pair. Unlike in the constant dependence distance case, here the dependence distances vary with the loop indices. Dependence distance is function of loop indices in this case. Thus, instead of summarizing the dependence for the entire loop, the summarization is done at each apex.

Figure 2.2: Variable dependence distances



As shown in the figure 2.2, apex1 is the initial apex point. We obtain apex2’ and apex2” as sinks of apex1. Then the sinks of all sources inside the cone of apex1 lie inside the cones formed by apex2’ and apex2”. We summarize these two apexes to obtain apex2 as the next apex point in the partition. This procedure is repeated to obtain subsequent apex points. Once the apex points are known, the partitions are created in similar way as was done in the constant dependence distance case.

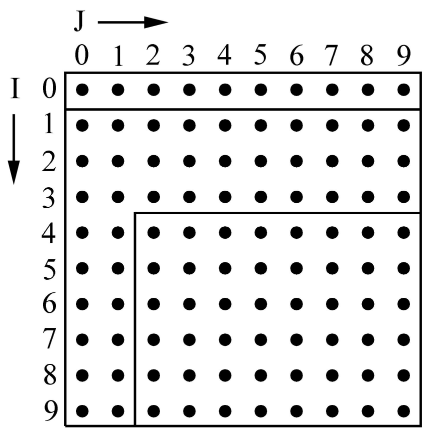
Consider the following example:

```
DO I = 0 to 9
    DO J = 0 to 9
        A[2I+J+3, I+4J+2] = B[I+3, J+3]
        B[3I+J+4, 2I+2J+3] = A[I+1, J+1]
    ENDO
ENDO
```

Dependence distances are : {(I+J+2, I+3J+1), (2I+J+1, 2I+J)}

apex1 = (0, 0)
Dependence distances: {(2, 1), (1, 0)}
Summarized dependence distance = (1, 0)
apex2 = (1, 0)

Figure 2.3: Partitions created in the example of variable dependence distance



Dependence distances: {(3, 2), (3, 2)}
Summarized dependence distance = (3, 2)
apex3 = (4, 2)

Dependence distances: {(8, 11), (11, 10)}
Summarized dependence distance = (8, 10)
apex4 = (12, 12) which is out of range and thus all partitions have been found

The partitions generated from this example are shown in the figure 2.3

Chapter 3

Analysis of Extended Cycle Shrinking Algorithm

A dependence distance $\Phi(R)$ is called a dominating dependence distance if $\Phi(L) = \Phi(R)$. A dominating dependence distance may not always exist.

The extended cycle shrinking algorithm is optimal only when there exists a dominating dependence distance. If cycle shrinking is applied to loops where this is not the case then it gives suboptimal partitions.

For example consider the following code:

```
DO I = 0, 9
    DO J = 0, 9
        A[I+1, J+9] = B[I, J]
        C[I+6, J+6] = A[I, J]
        B[I+8, J+1] = C[I, J]
    ENDO
ENDO
```

The dependence distances are (1, 9), (6, 6) and (8, 1). $\Phi(L) = (1, 1)$. Thus, there is no dominating dependence distance in this case. The partitions created by extended cycle shrinking algorithm and a better possible partition is shown in Figure 3.1. This motivates us to optimize the extended cycle shrinking algorithm so to have optimal partitions even in those cases where there is no dominating dependence distance.

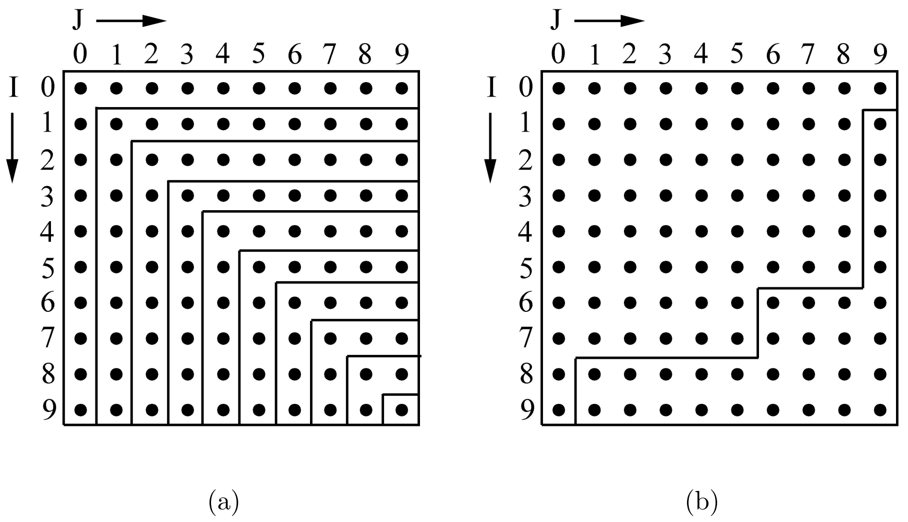


Figure 3.1: (a) Partitions by extended cycle shrinking algorithm. (b) Better partition

Chapter 4

Optimal Partitioning Algorithm

Consider the case of constant dependence distances. Consider only those cases where all the dependence distances are positive. The algorithm can then be easily extended when all dependence distances are negative for a particular loop nest.

4.1 Notation

For a nested loop, the vector of loop index for each nest is called as an iteration point. For example, consider a doubly nested loop with outer loop index i and inner loop index j. Then iteration point a = (2, 3) means that at this point, i = 2 and j = 3. Also $a_1 = 2$, $a_2 = 3$.

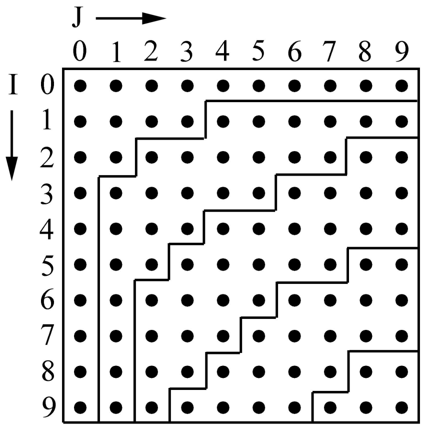
Iteration point a is said to dominate iteration point b if
$\forall k, a_k \leq b_k$
We denote it by $a \preceq b$.
This creates a partial order on the set of iteration points.

We denote a partition by two sets of points, set A and set B, belonging to the iteration space. This partition is represented by P(A,B).

P(A,B) is the set of all iteration points that are dominated by at least one iteration point in set A and are not dominated by any iteration point from set B
$P(A, B) = \{x | (\exists a \in A, a \preceq x) and (\forall b \in B, not(b \preceq x))\}$

Thus if we have sets $A_0, A_1, ...A_k$ then we have k partitions:
$P(A_0, A_1), P(A_1, A_2), P(A_2, A_3), ..., P(A_{k-1}, A_k)$.

Figure 4.1: Partitions by our algorithm



4.1.1 Example

For the example in figure 4.1, the sets are as given below:
$A_0 = \{(0,0)\}$
$A_1 = \{(1,4),(2,2),(3,1)\}$
$A_2 = \{(2,8),(3,6),(4,4),(5,3),(6,2)\}$
$A_3 = \{(5,8),(6,6),(7,5),(8,4),(9,3)\}$
$A_4 = \{(8,8),(9,7)\}$
$A_5 = \{\}$

4.2 Algorithm to generate the sets of points

We give an iterative algorithm in which, given a set of points, it generates the next set. Let the sets be called $A_0, A_1, .., A_k$

Consider the set of all the dependence distances of the loop. Create a minimal set of dependence distances by removing any dependence distance that is dominated by another dependence distance in the set. Thus in the minimal set, no dependence distance dominates any other dependence distance. Call this set D.

```
$A_0 = \{(0,0,..,0)\}$ /* Vector containing as many 0s as the total loop nest.
First element corresponds to outermost loop and so on. */
$iter = 0$
while($A_{iter}$ is not empty) {
    iter++
```

Table 4.1: Iterations of the algorithm applied to example

iter	**B**	A_{iter}
0	-	$\{(0,0)\}$
1	$\{(1,4),(2,2),(3,1)\}$	$\{(1,4),(2,2),(3,1)\}$
2	$\{(2,8),(3,6),(4,5),(3,6),(4,4),(5,3),(4,5),(5,3),(6,2)\}$	$\{(2,8),(3,6),(4,4),(5,3),(6,2)\}$
3	$\{(5,9),(5,8),(6,7),(5,8),(6,6),(7,5),(6,7),(7,5),(8,4),(7,6),(8,4),(9,3)\}$	$\{(5,8),(6,6),(7,5),(8,4),(9,3)\}$
4	$\{(8,9),(8,8),(9,7),(8,9),(9,7),(9,8)\}$	$\{(8,8),(9,7)\}$
5	$\{\}$	$\{\}$

```
    $B = \{\}$
    $\forall a \in A_{iter-1}, \forall d \in D$, insert a+d in B if a+d lies inside the loop bounds
    Create a minimal set of B, by removing an iteration point from B if there exists another iteration point dominating it.
    Thus in the final set obtained no iteration point dominates another point.
    Set $A_{iter}$ as this minimal set obtained
}
```

4.2.1 Example

For example consider the following code:

```
DO I = 0 to 9
    DO J = 0 to 9
        A[I+1, J+4] = B[I, J]
        C[I+2, J+2] = A[I, J]
        B[I+3, J+1] = C[I, J]
    ENDO
ENDO
```

In this case,
$D = \{(1,4),(2,2),(3,1)\}$
The above algorithm applied to this example is shown in table 4.1.
The partitions generated are depicted in figure 4.1.

4.3 Proof of Correctness

We need to prove that a source-sink pair can never lie in the same partition. We prove this by contradiction.

Suppose there exists a source u and sink v that lie in the same partition P(A,B). Let the dependence distance between them be d.
Thus $v = u + d$
Consider $d' \preceq d$ and $d' \in D$

Lemma 1: There always exists such d'

Let $v' = u + d'$
Thus, u, v' is another source-sink pair

Lemma 2: If $u, v \in P(A, B)$ then $u, v' \in P(A, B)$.

As, $u, v' \in P(A, B), \exists a \in A, a \preceq u$
By the method in which partitions are created,
$\exists b \in B, b \preceq a + d'$
$a \preceq u$
$\Rightarrow a + d' \preceq u + d'$
$\Rightarrow b \preceq a + d' \preceq v'$
$\Rightarrow b \preceq v'$
Thus it implies v does not belong to P(A,B) which is a contradiction.

Thus, our assumption is false. Therefore there does not exist any source-sink pair belonging to same partition.

4.3.1 Proof of Lemma 1

If $d \in D$ then $d = d'$
else as d was removed from the set of dependence distances, there must exist another dependence distance in D dominating d. Let that be d'

4.3.2 Proof of Lemma 2

$u, v \in P(A, B)$
$\Rightarrow \exists a \in A, a \preceq u$
$\Rightarrow a \preceq u \preceq v'$
$\Rightarrow a \preceq v'$

Suppose $v' \notin P(A, B)$.
$\Rightarrow \exists b \in B, b \preceq v'$
$d' \preceq d$
$\Rightarrow d' + u \preceq d + u$
$\Rightarrow v' \preceq v$
$(b \preceq v'$ and $v' \preceq v) \Rightarrow b \preceq v$
$\Rightarrow v \notin P(A, B)$
Therefore our assumption is false.
$v' \in P(A, B)$

4.4 Proof of Optimality

We need to prove that even if a single iteration point from a partition is added to its previous partition, it will lead to at least one source-sink pair belonging to same partition.

Suppose the partitions we created are not optimal. Therefore there exists a point y which is in the next partition of P(A,B) which can be added to P(A,B). i.e y has no source in P(A,B).

$y \notin P(A, B) \Rightarrow \exists b \in B, b \preceq y$
By the method in which partitions are created,
$\exists d \in D$ and $a \in A, a + d = b$
Let $x = y - d$
$b \preceq y \Rightarrow b - d \preceq y - d$
$\Rightarrow a \preceq x$
This means that $x \in P(A, B)$
x and y form a source-sink pair. Therefore there is a source of y in P(A,B).
Therefore our assumption is false.

Thus partitions created are optimal.

Chapter 5

Method to calculate the number of partitions

We may need to find the number of partitions required in this algorithm without actually creating partitions so as to get an idea of how much parallelization is possible in the loop. The method for that is given below.

Let the loop bounds be from 0 to $(n_i - 1)$ for loop nest i.
Let $d^1, d^2, ..., d^k$ be the elements of set D

The optimal value of the following integer linear optimization problem gives the number of partitions:

maximize $x_1 + x_2 + .. + x_k + 1$
subject to
$\Sigma_{j=1}^{k} d_i^j * x_j < n_i$ for each loop nest i
$x_j \geq 0 \; \forall j$
All x_j are integral.

By giving the above linear optimization problem to a solver, the optimal value caluclated is the number of partitions required.

5.0.1 Example

Consider the same example,

```
DO I = 0 to 9
    DO J = 0 to 9
        A[I+1, J+4] = B[I, J]
        C[I+2, J+2] = A[I, J]
        B[I+3, J+1] = C[I, J]
    ENDO
```

ENDO

$d^1 = (1, 4)$
$d^2 = (2, 2)$
$d^3 = (3, 1)$
$n_1 = 10$
$n_2 = 10$

The linear optimization problem for this case is:
maximize $x_1 + x_2 + x_3 + 1$
subject to
$1 * x_1 + 2 * x_2 + 3 * x_3 < 10$
$4 * x_1 + 2 * x_2 + 1 * x_3 < 10$
$x_1 \geq 0$
$x_2 \geq 0$
$x_3 \geq 0$
x_1, x_2, x_3 are integral.

Chapter 6

Loop Generation Algorithm

We have given an algorithm to find the sets of points that can determine the optimal partitions. Now we give an algorithm to generate actual loops given these sets of points.

6.1 Notation

generate_loop() takes two sets of points A and B as input and outputs the loops representing P(A, B)

A, B are ordered sets of points. The ordering is based on ascending order of first dimension of the points. A.first() gives the first point in the ordered set A. Point is a vector of indices.

A.insert(a) inserts the point a in the ordered set A. A.remove() removes the first point from set A.

remove_first(a) removes the first dimension from the vector. For example, remove_first([1,2,3,4]) returns [2,3,4].

dominate(A) removes redundant points from A. All points in set A that are dominated by some other point in A are removed from A. i.e.,
if ($a, b \in A$ and $a \preceq b$) then b is removed from A.

boundaries is a vector of upper bound of loop indices.

l denotes the current loop nest level. It should be initially called with 0.

6.2 Algorithm

```
generate_loop (A, B, boundaries, l) {

    A = dominate(A)
    B = dominate(B)

    //BASE CASE
    if A and B have points with only 1 dimension then {
        start = A.first()[0];
        if (B is empty) {
            end = boundaries[l];
        }
        else {
            end = B.first()[0] - 1;
        }
        put FORALL $i_l$ = start to end
        put statements block inside the loop to be executed
        put ENDALL

        return;
    }

    A' = {}
    B' = {}
    start = min (A.first()[0], B.first())

    put PARBEGIN statement

    while (A is not empty or B is not empty) {

        while (A.first()[0] == start) {
            A'.insert(remove_first(A.first()))
            A.remove()
            // eg: if first point in A is (2, 4, 1), then put (4, 1) in A'
        }
        while (B.first()[0] == start) {
            B'.insert(remove_first(B.first()))
            B.remove()
        }

        if (both A and B are empty) {
            end = boundaries[l];
        }
        else {
            end = min(A.first()[0], B.first()[0]) - 1;
        }

        put FORALL $i_l$ = start to end
        generate_loop (A', B', boundaries, l+1);
        put ENDALL

        start = end + 1
    }
    put PAREND statement
}
```

6.3 Example

Consider the following example:

```
DO I = 0 to 9
    DO J = 0 to 9
        DO K = 0 to 9
            {Block of statements}
        ENDO
    ENDO
ENDO
```

In this case, boundaries = (9, 9, 9)
Let A = {(1, 4, 1), (5, 1, 3)} and B = {(4, 6, 7), (7, 2, 5)}

generate_loop(A, B, boundaries, 0) generates the following code:

```
PARBEGIN
FORALL I = 1 to 3
    PARBEGIN
    FORALL J = 4 to 9
        FORALL K = 1 to 9
            {Block of statements}
        ENDALL
    ENDALL
    PAREND
ENDALL
FORALL I = 4 to 4
    PARBEGIN
    FORALL J = 4 to 5
```

```
        FORALL K = 1 to 9
            {Block of statements}
        ENDALL
    ENDALL
    FORALL J = 6 to 9
        FORALL K = 1 to 6
            {Block of statements}
        ENDALL
    ENDALL
    PAREND
ENDALL
FORALL I = 5 to 6
    PARBEGIN
    FORALL J = 1 to 3
        FORALL K = 3 to 9
            {Block of statements}
        ENDALL
    ENDALL
    FORALL J = 4 to 5
        FORALL K = 1 to 9
            {Block of statements}
        ENDALL
    ENDALL
    FORALL J = 6 to 9
        FORALL K = 1 to 6
            {Block of statements}
        ENDALL
    ENDALL
    PAREND
ENDALL
FORALL I = 7 to 9
    PARBEGIN
    FORALL J = 1 to 1
        FORALL K = 3 to 9
            {Block of statements}
        ENDALL
    ENDALL
    FORALL J = 2 to 3
        FORALL K = 3 to 4
            {Block of statements}
        ENDALL
    ENDALL
    FORALL J = 4 to 9
        FORALL K = 1 to 4
```

```
            {Block of statements}
        ENDALL
    ENDALL
    PAREND
ENDALL
PAREND
```

6.4 Proof of Correctness

6.4.1 Notation

[a:b] denotes the vector obtained on inserting element a at front of vector b. For example [1:(2, 3, 4)] denotes (1, 2, 3, 4)

6.4.2 Lemma1

$(a > b) \implies (\forall x, \forall y, not([a : x] \preceq [b : y]))$
Proof is straightforward from definition of $\preceq$

6.4.3 Lemma2

$(a \preceq b$ and $x \leq y) \implies ([x : a] \preceq [y : b])$
Proof is straightforward from definition of $\preceq$

6.4.4 Lemma3

$(not(a \preceq b)) \implies (\forall x, \forall y, not([x : a] \preceq [y : b]))$
Proof is straightforward from definition of $\preceq$

6.4.5 Proof

We need to prove that the set of points in P(A, B) and those generated by this algorithm are same.
Here $P(A, B) = \{x | (\exists a \in A, a \preceq x) and (\forall b \in B, not(b \preceq x))\}$

The proof is based on induction on the dimension of the points.

Base case (dimension = 1):

When dimension is one, there can be exactly one point (say a) in set A and at most one point (say b) in B after removing redundant points. Thus, P(A, B) contains iteration points from a to b-1 when b exists. Otherwise it contains iteration points from a to boundary. The same iteration points are included by the algorithm also. Thus base case is proved.

Induction case (dimension = k):

For each value of x,
$\forall a \in A, (first(a) > x) \implies (\forall y, not(a \preceq [x : y]))$ (By Lemma1).
Therefore those a where $first(a) > x$ need not be considered.

We have $A' = \{a \in A | first(a) \leq x\}$.
By induction the partition will include those y where $\exists a' \in A'$ such that $a' \preceq y$

($\forall a$ such that its corresponding $a' \in A', first(a) \leq x$ and $\exists a' \in A', a' \preceq y) \implies (\exists a \in A, a \preceq [x : y])$ (by Lemma2).

$\forall b \in B, (first(b) > x) \implies (\forall y, not(b \preceq [x : y]))$ (by Lemma1).
Therefore, those b where $first(b) > x$ satisfy the condition and need not be considered.

$B' = \{b \in B | first(b) <= x\}$.
By induction, it will include only those points y where $\forall b' \in B', not(y \preceq b')$

$not(y \preceq b') \implies not([x : y] \preceq [first(b) : b'])$ (by Lemma 3).

Thus, $\forall b \in B, not([x : y] \preceq b)$.

Thus, we have included those iteration points [x:y] where $\exists a \in A, a \preceq [x : y]$ and $\forall b \in B, not(b \preceq [x : y])$. Thus, all the points included by the algorithm is same as the set of points defined by P(A,B).

Chapter 7

Variable Dependence Distance

We have given an optimal algorithm for the constant dependence distance case. Now we will consider variable dependence distances.

In the case of variable dependence distances, we consider only those cases where each source can have only one sink for a given reference pair. As the dependences vary with the loop indices, dependence distance is a function of loop indices. Thus, instead of having a constant set of dependence distances, we will calculate the dependence distances at each point where required.

7.1 Algorithm

We give an iterative algorithm in which, given a set of points, it generates the next set. Let the sets be called $A_0, A_1, .., A_k$

Consider the set of all the dependence distances of the loop. Let this set be D. Note that the dependence distances in D depend on loop indices. For a dependence distace d, d(a) represents the distance when calculated at iteration point a.

```
$A_0 = \{(0, 0, .., 0)\}$ /* Vector containing as many 0s as the total loop nest. First element corresponds to outermost loop and so on. */
$iter = 0$
while($A_{iter}$ is not empty) {
    iter++
    $B = \{\}$
    $\forall a \in A_{iter-1}, \forall d \in D$, insert a+d(a) in B if a+d(a) lies inside the loop bounds
    Create a minimal set of B, by removing an iteration point from B if there
```

Table 7.1: Iterations of the algorithm applied to example

iter	**B**	A_{iter}
0	-	$\{(0,0)\}$
1	$\{(2,0),(1,3)\}$	$\{(2,0),(1,3)\}$
2	$\{(6,2),(3,5)\}$	$\{(6,2),(3,5)\}$
3	$\{\}$	$\{\}$

Table 7.2: Dependence distance calculation

a	**D(a)**	**a + D(a)**
(0,0)	{(2,0), (1,3)}	$\{(2,0),(1,3)\}$
(2,0)	{(4,2), (1,5)}	$\{(6,2),(3,5)\}$
(1,3)	{(6,7), (4,7)}	$\{(7,10),(5,10)\}$
(6,2)	{(10,10), (3,11)}	$\{(16,12),(9,13)\}$
(3,5)	{(10,13), (4,11)}	$\{(13,18),(7,16)\}$

```
exists another iteration point dominating it.
    Thus in the final set obtained no iteration point dominates another point.
    Set $A_{iter}$ as this minimal set obtained
}
```

Once the sets $A_0, A_1, .., A_k$ are generated, the partitions obtained are $P(A_0, A_1), P(A_1, A_2), ..., P(A_{k-1}, A_k)$. The algorithm for loop generation from these sets of points is same as in the case of constant dependence distances.

7.2 Example

Consider the following example:

```
DO I = 0 to 9
    DO J = 0 to 9
        A[2I+J+3, I+3J+1] = B[I+3, J+3]
        B[I+J+4, I+2J+6] = A[I+1, J+1]
    ENDO
ENDO
```

D = {(I+J+2, I+2J), (J+1, I+J+3)}

The algorithm applied to this example is shown in the table 7.1 and table 7.2. The partitions created are shown in the figure 7.1a.

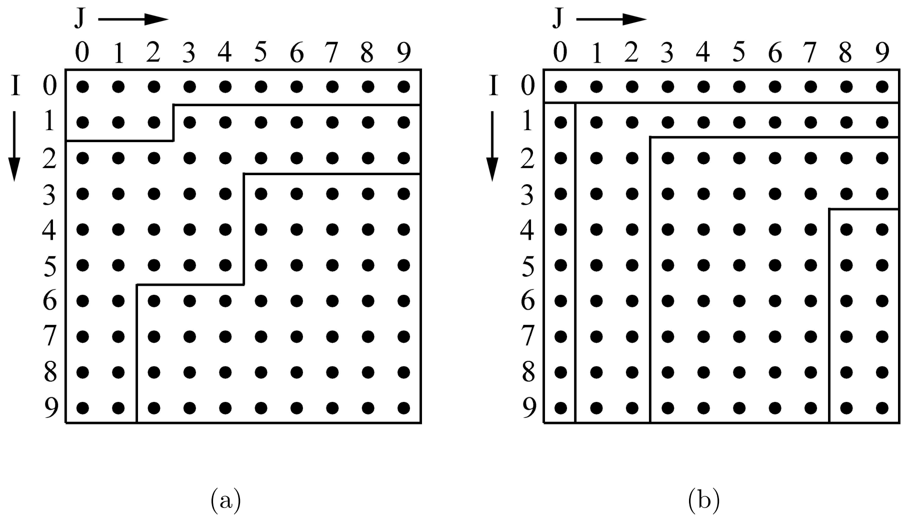


Figure 7.1: (a) Partitions created by our algorithm. (b) Partitions created by extended cycle shrinking algorithm

7.3 Optimality of number of partitions created

Now we analyse the optimality of the partitions thus created by our algorithm.

7.3.1 Comparison with Extended Cycle Shrinking Algorithm

For the example above, the partitions generated are shown in the figure 7.1. Our algorithm generates only 3 partitions whereas extended cycle shrinking algorithm generates 5 partitions in this example. Thus in this case, our algorithm is better than the extended cycle shrinking algorithm. The main reason for this is that extended cycle shrinking algorithm summarizes the distances whereas our algorithm doesn't summarize. Thus, the benefit obtained here is similar to the benefit obtained in the case of constant dependence distances.

Even though the partitions generated by our algorithm are not optimal in the sense that more iteration points could have been added to a partition, we can prove that the number of partitions created is optimal. The proof of this is presented below.

7.3.2 Proof of optimality

Consider the sets of points $A_0, A_1, ..., A_k$ generated by the algorithm representing the k partitions, $P(A_0, A_1), P(A_1, A_2), ..., P(A_{k-1}, A_k)$.

Consider a point $a_{k-1} \in A_{k-1}$.
By the method in which partitions are created,
$\exists a_{k-2} \in A_{k-2}, a_{k-2} \Delta a_{k-1}$. i.e., a_{k-2} and a_{k-1} are source and sink of a dependence. Similarly, we can obtain $a_{k-3} \in A_{k-2}, a_{k-3} \Delta a_{k-2}$ and so on till we obtain a_0.

Now consider the k points $a_0, a_1, ..., a_{k-1}$ obtained as above.
$a_0 \Delta a_1$ implies a_0 and a_1 must belong to difference partitions.
Similarly, $a_1 \Delta a_2$ implies a_1 and a_2 must belong to different partitions and so on.

Thus, each of $a_0, a_1, ..., a_{k-1}$ must belong to different partitions. Therefore in an optimal partitioning, there will be at least k partitions. But our algorithm created k partitions.

Thus, the number of partitions created are optimal.

Chapter 8

Handling negative dependence distances

Till now we only considered the cases where dependence distances along all indices are postive. We can easily extend our algorithm to work in cases when dependence distances have negative values along some indices.

If along a particular index, some dependence distances have positive values whereas some have negative values, then we summarize the dependence distance to 0 along that index in all the dependence distances

If along some index, all dependence distances have negative values then we convert all of them to positive but remember the information that that index has negative dependence distance value. Then along that index, we replace the occurence of that index with (boundary - index) in all the statements and generate the code as in the case of positive dependence distances.

This substituting of J with (boundary - J) implicitly makes the traversing of index J in reverse direction. This handles the negative dependence distance cases.

Chapter 9

Parallelizing the Algorithm

We now give some ideas regarding parallelizing our cycle shrinking algorithm.

Consider the algorithm for generating partitions in the constant dependence case. Our algorithm uses previous set of points to generate the next set of points. But instead, we can directly generate the n^{th} set of points using the following method:
$B_n = \{b | b = x_1 d_1 + x_2 d_2 + ... + x_m d_m, x_1 + x_2 + ... + x_m = n, \forall i, x_1 \geq 0\}$
where $d_1, d_2, ..., d_n$ are the dependence distances.
$A_n = dominate(B_n)$

Now with this method, all sets of points can be calculated in parallel as they don't depend on each other. We can find the number of sets required beforehand by our method to calculate the number of partitions.

For generating the loops from set of points $A_0, A_1, ..., A_k$, we need to generate loops for $P(A_0, A_1), P(A_1, A_2), ..., P(A_{k-1}, A_k)$. All of them can be done in parallel as they do not use any information from others.

Chapter 10

Conclusion and Future Work

For the constant dependence distance case, we have given the algorithm to generate the optimal partitions possible and for the variable dependence distance case also we have shown that the algorithm generates the optimal number of partitions theoretically. These can be taken up further in order to realise this on an architecture and actually make use of optimal partitions.

The complete algorithm can be implemented. A testing platform can be created and the algorithm can be put to test so as to have performance analysis of the algorithm.

Bibliography

[1] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures. 2002.* Elsevier Pub.

[2] C.D. Polychronopoulos. “Compiler optimizations for enhancing parallelism and their impact on architecture design”. In: *Computers, IEEE Transactions on* 37.8 (1988), pp. 991–1004. ISSN: 0018-9340. DOI: `10.1109/12.2249`.

[3] AJAY SETHI, SUPRATIM BISWAS, and AMITABHA SANYAL. “EXTENSIONS TO CYCLE SHRINKING”. In: *International Journal of High Speed Computing* 07.02 (1995), pp. 265–284. DOI: `10.1142/S0129053395000154`.