Analysis of Loops with Variable Distance Data Dependence for Parallelization

Third Annual Progress Seminar Report

by

Archana Kale
Roll No: 114058002

under the guidance of

Prof. Supratim Biswas

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai

Abstract

Parallelizing sequential program code requires data dependence analysis. Loops, which are often used for processing arrays, are iterative and consume many machine cycles. Parallelizing loops of sequential program is necessary for parallelization, and as a result requires data dependence analysis. There is data dependence between two or more references to the same location in same or different iterations, if one of them is a write access. Data dependence between iterations restricts parallelization of loops. Data dependence distance is the measure of difference between two dependent iterations. When dependence exists between two iterations, the relative order of execution of iterations needs to be maintained to preserve the program semantics. When the distance between two iterations is constant it is called constant distance. When this distance varies for different pairs of iterations it is called variable distance. Best performance of many-core or multi-core architectures is achieved by maximizing parallelism. This requires partitioning with largest number of components by obtaining correct data dependence information. Constant distance data dependence has been studied in literature and implemented in compiler passes. However, some literature is available for parallelization in the case of variable distance data dependence(VD^3).

To study VD^3 in greater details across iterations is the focus of this work. It is our belief that partitioning the iteration space in such loops cannot be done without examining solutions of the corresponding Linear Diophantine Equations(LDEs). Focus of this work is to study VD^3 and examine the relation between dependent iterations. Analysis based on parametric solution, leading to a mathematical formulation capturing dependence between iterations was presented in previous Annual Progress Seminar(APS). Our approach shows the existence of reasonable exploitable parallelism in VD^3 loops with multiple LDEs. The work done this year includes noticeable improvement in exploitable parallelism using heuristic based splitting. This work extends solutions of 2 variable LDEs to solutions of multi variable LDEs. Solutions of LDEs having multiple variables corresponding to multiple array access nested in multiple loops are presented in this report. The precise dependence thus obtained is presented and compared with some earlier work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Loops are iterative code segments, frequently used for processing arrays. During program execution, loops consume many machine cycles. Due to this, parallelizing loops is necessary for parallelization of sequential programs. The extent of parallelization of a loop is largely determined by the dependences between its statements. While dependence free loops are fully parallelizable, those with loop carried dependences are not. Dependence distance is a measure of absolute difference between a pair of dependent iterations.

Loops with constant distance data dependence(CD^3) have uniform distance between the dependent iterations. This leads to easy partitioning of the iteration space and hence they have been successfully dealt with in literature[**?**][**?**] constant. Loops with loop carried CD^3s having cycle free Data Dependence Graph(DDG) have been successfully dealt with in the literature[**?**]. A few researchers have also considered loops having cycles in DDG[**?**, **?**].

Loops with variable distance data dependence(VD^3) have non-uniform distance between the dependent iterations. Some efforts have considered VD^3 loops and DDGs with or without cycles[**?**, **?**, **?**, **?**, **?**]. Unlike in the case of CD^3, the solutions of LDEs for VD^3 do not seem to have a regular structure among dependent iterations. Parallelization of loops with VD^3 is a considerably difficult problem. Partitioning the iteration space in such loops will require examining solutions of the corresponding Linear Diophantine Equations(LDEs). Focus of this research is to study VD^3 and examine the relation between dependent iterations.

1.1 Summary of Previous Work

During previous Annual Progress Seminar(APS), analysis based on well known parametric solution[**?**] leading to a mathematical formulation capturing dependence between iterations was presented for a 2 variable LDE. We had analyzed the parametric solutions and had developed a mathematical formulation that captured the structure among dependent iterations. The structure led to partitioning of the iteration space where each connected component(CC) of the partition contained iterations which had to be executed sequentially and distinct components were parallelizable. Further, bounds on the number and size of CCs had been obtained, and iterations of a CC were generated from a single iteration, called its Seed. The representation of the partition consequently reduced to a set of Seeds (1 per CC) in place of all source-sink pairs. It was observed that these partitions make such loops amenable to parallelism.

Further, the mathematical model for partitioning of 2 variable LDEs was generalized for integer or non-integer values of loop independent iteration α. A test for independent iterations was designed. Heuristic based loop splitting was presented which noticeably enhanced the exploitable parallelism in case of 2 variable LDEs. Building over generalized solutions of 2 variable LDEs solutions for multi-variable LDEs were obtained in a novel way.

Solutions thus obtained were merged to obtain solution vectors of complete solution. We obtained the complete solution in a manner similar to generation of final direction vector for a loop from individual direction vectors obtained for different loop indices. We used cross product to generate complete solution over solutions of modified LDEs for different loop indices. Further, Solutions of multiple, multi-variable LDEs were obtained by merging solutions of single multi-variable LDEs depending on access patterns. The array accesses were categorized as separable, partially separable and non-separable array access and separable array accesses were handled.

The directions for future given last year were: computing tighter bounds for Fmerge(now represented by operator $\uplus$), finding a suitable splitting method which may or may not use heuristic and evaluate effect of splitting on multiple multi variable LDEs, extending this approach for loops with trapezoidal bounds, extending Precise dependence analysis, its resultant partition and exploitable parallelism to Partially separable and Non-separable array accesses and Generating efficient schedules using precise dependence analysis.

1.2 Referee's Comments on Previous Work

We had submitted a paper on the research work done to International conference on Super Computing 2014. The referee's officiated the contributions of the paper as:

From a technical perspective, the beginning of the paper was well written, the paper explained the general concept and presented the mathematical argumentation. The paper was built on a solid foundation and provided new insights into loop optimization, and the authors presented their research in a way one could relate to it and see its potential.

The following is the gist of improvements recommended by the referee's: The authors should highlight the main accomplishments of the paper, ie the "Precise dependence" and the Fmerge algorithm. Add some examples and/or visualizations, describe existing examples in more detail. Moreover, the authors should expand Comparative Analysis and explain the meaning of the results. Improvement in language for better readability was recommended and justification for use of artificially constructed loops was desired.

The work done this year includes modifications made after taking a serious not of referee's comments. The following section highlights modifications and extensions made in the approach this year.

1.3 Modifications and Extensions made in the approach this year

During experimentation we found that the parallelization was subjective to the context of LDEs. Due to this LDEs were categorized based on:

- LDEs corresponding to single dimensional array access, having single loop index in its subscripts

- LDEs corresponding to single dimensional array access, having multiple loop indices in its subscripts
- LDEs originating from different subscript positions of a pair of array access.
- LDEs corresponding to access of multiple arrays in the nested loop.

The work done this year categorizes different types of LDEs. This categorization helped in improving clarity and readability. This led to further realization that LDEs of each type had to be merged differently. Nomenclature of heuristic based transformation was changed from loop distribution to loops splitting. Spitting is a standard term and correctly describes the transformation.

Earlier efforts used cross product to generate complete solution over solutions of modified LDEs for different loop indices. This approach generated the required solutions along with some spurious results. To obtain precision of results, the merging methods were improved. Solutions of multiple, multi-variable LDEs are obtained by merging solutions of single multi-variable LDEs depending on access patterns. While one merging method uses intersection, the other involves a nontrivial merging using pair wise intersection between CCs. These methods are explained in **??** and **??**.

Precise dependence analysis its and resultant partition was extended to Partially separable and Non-separable array accesses. The results of comparative analysis made on previous work[?, ?, ?] were reworked as the method improved.
The section 1.4 presents a motivating example of a loop structure for parallelization.

1.4 Motivating Example

Presented here is an example of a loop nest having different types of array access. The categories of different types of LDEs depending on the context and type of array accesses is shown here.

```
for( i = 0; i < 100 ; i++)
{
    b[2i + 1] = ....;
    ......... = b[3i + 6];
    for( j = 0; j < 100 ; j++)
    {
        a[2i + 1][2j + 4] = c[i + 2j];
        c[2i + 3j + 6] = d[2i +  j ][i + j + 5];
        d[i + 4][3i + 2j]= a[i][3j];
    }
}
```

The afore mentioned example shows 2 loops with 2 loop induction variables i corresponding to outer loop and j corresponding to inner loop used for accessing arrays a,b,c and d. The categorization of LDEs depending on the context of access with respect to data dependence analysis is:

1. **Type 1(2 variable LDE):** The dependence relation corresponding to single dimensional array access, having single loop index in its subscripts is represented by a 2 variable LDE. The dependence relation corresponding to single dimensional array **b** is represented by LDE : $2x_1 - 3y_1 = 5$.

2. **Type 2(multi variable LDE):** The dependence relation corresponding to single dimensional array access, having multiple loop indices in its subscripts is represented by multi variable LDE. The dependence relation corresponding to single dimensional array **c** is represented by 4 variable LDE : $x_1 - 2y_1 + 2x_2 - 3y_2 = 6$. In this LDE x_1, y_1 correspond to induction variable i and x_2, y_2 correspond to induction variable j.

3. **Type 3(System of LDEs):** Multiplicity in either dimension or number of array accesses is represented by a system of LDEs.

   (a) **3a:** The dependence relation corresponding to multidimensional array access in the nested loop is represented by one LDE per dimension, forming a system of LDEs.The dependence relation corresponding to 2 dimensional array **a** is represented by a system of 2 LDEs : $2x_1 - y_1 = -1$ and $2x_2 - 3y_2 = -4$. In this system of LDEs x_1, y_1 correspond to outer loop induction variable i and x_2, y_2 correspond to inner loop induction variable j.

   (b) **3b:** The dependence relation corresponding to reference pairs of multiple arrays in the nested loop is also represented by a system of LDEs. These arrays may belong to Type 1, Type 2 or Type 3a categories stated above.The dependence relation corresponding to 2 dimensional array **d** is represented by a system of 2 LDEs : $2x_1 - y_1 + x_2 = 4$ and $x_1 - 3y_1 + x_2 - 2y_2 = -5$. In this system of LDEs x_1, y_1 correspond to outer loop induction variable i and x_2, y_2 correspond to inner loop induction variable j.

Precise Dependence(PD): Given an Iteration space and a set of LDEs, we use the term Precise dependence to denote the collection of all solutions of all LDEs in the iteration space. Iterations which do not appear in PD are independent iterations. PD captures all the parallelism present at iteration level granularity.

1.5 Overview of the Report

Chapter 1, Introduction brings forward the need of VD^3. The summary of earlier work, modifications, extensions and contributions of this year are presented. The **Categorization of LDEs** justifying its need is presented in section 1.4.

Solution of 2 variable LDE is presented in Chapter 2. This includes mathematical formulation of partition **P**, its **CCs** and **Seeds**. Section 2.9 presents a **Test of independent iteration** and section 2.10 presents **Heuristic based splitting**.

Our work this year extends solutions of 2 variable LDE to solve multi variable LDEs and is reported in Chapter **??**. Section **??** presents the method of constructing LDEs from reference pairs of arrays in the loop nest. Solutions of multi variable LDE are obtained by merging CCs of solutions of system of 2 variable LDEs in a non trivial way by taking a pair of element of CCs at a time as shown in **??**. Multiple such solutions are merged by union for overlapping CCs. The computation is represented by operator $\uplus$ and shown in section **??** followed by solved examples in **??**.

In Chapter **??** an approach to solve system of LDEs i.e. multiple, multi variable LDEs is shown which uses solutions of different LDEs generated using methods mentioned in Chapters 2 and **??**. LDEs of subscripts are merged by intersection and that of different array accesses by $\uplus$. Our finds show that the composite solution depends on the overlap between components of constituent solutions. This is reported in Chapter **??** and explained with solved examples in section **??**. In general array accesses in loops can be called as **separable** array access if each loop induction variable appears in the subscript expression of exactly one specific dimension for all references of that array. Our method computes PD for all types of separabilities.

Chapter **??** presents earlier work done. Loops with loop carried CD^3s having cycle free Data Dependence Graph(DDG) have been successfully dealt with in the literature[**?**]. Researchers have considered loops having cycles in DDG[**?**] and VD^3 loops having DDGs with and without cycles [**?**]. Unlike the case of constant distance, the solutions of LDEs for VD^3 do not seem to have a regular structure among dependent iterations. Methods for handling VD^3 include approaches based on Uni-modular transforms[**?**], dependence analysis using recurrence chains[**?**] and a slicing based approach[**?**].

Heuristic based splitting is implemented for 2 variable LDEs and results are presented in Chapter 3. Heuristic based splitting largely enhances exploitable parallelism in VD^3 loops. The Chapter **??** presents comparison of our approach with earlier mentioned techniques.

The last chapter, Chapter 4 presents concluding remarks and plans for further work.

Chapter 2

Single loop with Single LDE

Work done in this Chapter is same as was presented earlier. The nomenclature of the transformation based on heuristic has been changed for correctness form distribution of loop to splitting of loop.

2.1 Motivating Example

Consider the following loop accessing a one dimensional array and Linear Diophantine Equation (LDE) : $2x - 3y = 5$ representing the underlined dependence relation.

```
for(i = -10 ; i < 21 ; i++)
{ S1 g[2i + 1] = .........;
  S2 ......... = g[3i + 6];}
```

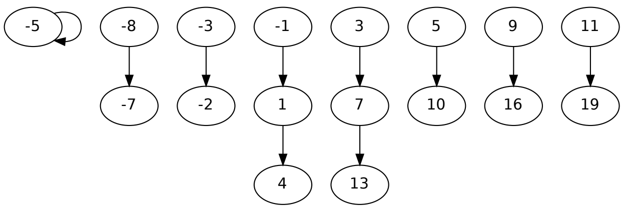

Figure 2.1: Dependent iterations of Loop (LDE $2x - 3y = 5$)

Figure 2.1 shows Dependent iterations of the Loop (represented by LDE). All the remaining iterations are Independent. The Independent and Dependent sets of iterations form Partition of the iteration space. A Connected Component (CC) of the Partition is a ordered relation of dependent iterations of the Loop. Iterations of different CCs are independent. Different CCs of LDE are $\{-8, 7\}, \{-5\}, \{-1, 1, 4\}, \{11, 19\}$ etc.
A Seed is a generator element of a CC. Following are subsets of Seeds of LDE $2x - 3y = 5$:

- Central = $\{-5\}$, is a Seed representing loop independent iteration.
- Left = $\{-8, -7\}$, are Seeds which precede Central in iteration order.
- Right = $\{-3, -1, 3, 5, 9, 11, ..\}$, are Seeds which succeed Central in iteration order.

- Independent $= \{-10, -9, -6, -4, 0, 2, 6, 8, 12, 13, 14, 15, 17, 18, 20\}$, are Seeds representing completely independent iterations.

A solution of LDE is a dependent iteration pair and the iterations are elements of the same CC. If an iteration i is common iteration of two solutions $\{k, i\}$ and $\{i, j\}$ then k, i and j belong to the same CC.
Union of CCs obtained from Central, Left and Right form S, set of solutions of LDE.
$Independent = R - S$.

The sections 2.2 to 2.6 report work presented in first APS including mathematical formulation of CCs and Seeds based on Normalized form of LDE. Computation of Bounds on length and number of CCs was presented. These bounds are characteristics of the Partition and are useful inputs for the scheduler. An Algorithm for generation of one Seed per CC was presented. This Chapter also presents work done this year from section 2.7 i.e. generalization for LDEs having integer / non-integer values of α and test for independent iterations.

2.2 Normalized form of LDE

An LDE which characterizes VD^3 can be normalized as:
The LDE is represented in its simplified form where, a, b, c are integer constants, $a > 0$ is coefficient of x, b is the coefficient of y, $gcd(a, b)$ divides c and $a \leq |b|$.
In case an LDE with $a > |b|$ exists, an equivalent normal form of LDE can be used. The Partition of an LDE $ax + by = c$ is the solution over its iteration space. If the iteration order is maintained, the CCs obtained are same as that of LDE $bx + ay = c$. If (g,h) is a solution of $ax + by = c$ then (h,g) is a solution of $bx + ay = c$. Hence it can be normalized to afore stated form for generation of Seeds and CCs.

$\alpha = c/(a + b)$, is a rational number if $a + b \neq 0$. If $a + b = 0$, alpha does not exist and such LDEs have CD^3.

The basic solution of an LDE is dependent on values of a,b and c. Slope of LDE is significant in computation of set of solutions. In the next section slopes of LDEs are used for categorizing CCs formed for different types of LDEs.

2.3 Formulation of CCs of LDE

This section presents different types LDEs, their CCs and computes upper bounds on number and length of CCs,N_{max} and L_{max}. R is the range of iteration space. The types of LDEs are:

A: LDE with $|slope| \neq 1$
LDEs used for analysis are represented in a reduced form where coefficient of x, $a > 0$ and $|slope| < 1$.

B: LDE with slope equal to -1
LDE's of this type are $x + y = c$. $\alpha = c/2$. CCs are: $\{(\alpha - k), (\alpha + k)\}$, where k is an integer if c is even and k + 0.5 is an integer if c is odd. $L_{max} = 2$ and $N_{max} = \frac{R}{2}$.

C: LDE with slope equal to 1
LDE's of this type $(x - y = c)$, have constant dependence distance. The CCs are of the

form $\{Seed, Seed + c, ..., Seed + mc\}$ bound by [LB,UB]. If $c = 0$, then all iterations are loop independent, $L_{max} = 1$ and $N_{max} = R$. If $c \neq 0$, α does not exist, $L_{max} = \frac{R}{c}$ and $N_{max} = c$.

The afore-listed cases completely cover the 1 dimensional iteration space for single loop. Type A is used for further analysis, B is special case of A. C is case of constant dependence and is handled separately.

2.4 Pattern of CCs for $ax + by = c$:

The pattern of CCs for an LDE using normalized form is presented in this section. (where coefficient of x, $a > 0$, coefficient of y is b, $|a| < |b|$, gcd(a,b) = 1 and loop independent iteration is α.)

Pattern of CC of length l=i+j+1, where m is mot a multiple of a or b are and $0 \leq i, j$ bound by [LB,UB]:

$\{(\alpha \pm ma^l), (\alpha \mp ma^{l-1}b), (\alpha \pm ma^{l-2}b^2), ..., (\alpha \pm (-1)^j.ma^ib^j), ...,$
$(\alpha \pm (-1)^{l-2}.ma^2b^{l-2}), (\alpha \pm (-1)^{l-1}.mab^{l-1}), (\alpha \pm (-1)^l.mb^l)\}$

A sub-CC is a CC which has an end node of the form $(\alpha \pm ma^ib^j)$.

Longer CCs join two smaller sub-CCs if m is a multiple of a or b, the length of new CC is sum of lengths of smaller CCs - 1.

Lemma 1: CCs of length 2 are given by $(\alpha \pm ma), (\alpha \mp mb)$.
Proof: $\{\alpha, \alpha\}$ is a particular solution to LDE. This is a CC of LDE. $\{(\alpha \pm ma), (\alpha \mp mb)\}$ are parametric solutions of LDE using particular solution. Hence $(\alpha \pm ma), (\alpha \mp mb)$ is the pattern for CCs of length 2 [?]. Hence Proved.

Lemma 2: Prove that longer CCs connecting sub-CCs are formed if m is a multiple of power of a or b.
Proof: Consider a solution such that CC = $\{(\alpha \pm ma), (\alpha \mp mb)\}$. If $m = ka$ is a multiple of a, element $(\alpha \pm kbb) \in$ CC by substituting $y = \alpha \pm ma$. If $m = kb$ is a multiple of b, element $(\alpha \mp kaa) \in$ CC by substituting $x = \alpha \mp mb$. This addition increases the length of CC. Hence proved.

Theorem 1: The general structure of CCs of LDE, where m is not a multiple of a or b is:
$\{(\alpha \pm ma^l), (\alpha \mp ma^{l-1}b), (\alpha \pm ma^{l-2}b^2), ..., (\alpha \pm (-1)^j.ma^ib^j), ...,$
$(\alpha \pm (-1)^{l-2}.ma^2b^{l-2}), (\alpha \pm (-1)^{l-1}.mab^{l-1}), (\alpha \pm (-1)^l.mb^l)\}$
Proof: Using Lemma(1) and Lemma(2).

Consider example of LDE $2x - 3y = 5$, shown in figure 2.1.

- The loop independent iteration $\alpha = -5$, $a = 2$, $b = -3$.
- A CC of length 2 is: $\{-5 - 1 * 2 = -7, -5 + 1 * (-3) = -8\}$ i.e. $\{-8, -7\}$.
- $\{-5 + 1 * 2^2 = -1, -5 - 1 * 2 * (-3) = 1, -5 + (-3)^2 = 4\}$
  i.e. $\{-1, 1, 4\}$ is a CC of length 3.
- -8 and -1 are seeds of the CCs.

2.5 Bounds on L_{max} and N_{max} for $ax + by = c$:

This section presents provable upper bounds on number and length of CCs.

2.5.1 Computation of Bounds

The sub-section shows bounds on number and length of CCs for LDE in a given range R = [L,U]

- Number of CCs of length l (including sub-CCs of length $< l$) $= \frac{R}{|b|^{l-1}}$.
- p_{max} is a positive integer such that $\frac{R}{|b|^{p_{max}+1}} < 1 \leq \frac{R}{|b|^{p_{max}}}$.
- The upper bound on length of CCs is $L_{max} = p_{max} + 1$.
- The upper bound on Number of CCs which forms an Arithmetic Progression is: $N_{max} = \frac{R}{|b|} - \frac{R}{|b|^2} + \frac{R}{|b|^3} - \frac{R}{|b|^4} \cdots \frac{R}{|b|^{p_{max}}} = N_{max} = R\Big(\frac{|b|^{p_{max}} - (-1)^{p}_{max})}{|b|^{p_{max}}.(|b|+1)}\Big)$.

2.5.2 Proofs of Bounds

Lemma 3: Number of CCs of length 2 is $\frac{R}{|b|}$.
Proof: Every solution of LDE is a CC of length 2. The number of solutions in the range R is $\frac{R}{|b|}$. Hence proved.

Lemma 4: Number of CCs of length l (including sub-CCs of length $< l$) $= \frac{R}{|b|^{l-1}}$.
Proof by Induction: Number of CCs of length 2 is $\frac{R}{|b|}$ using Lemma(3).
Assumption: Number of CCs of length k is $\frac{R}{|b|^{k-1}}$.
To Prove That: Number of CCs of length $k+1$ is $\frac{R}{|b|^k}$.
Last element of a CC of length k has a form $(\alpha \pm ma^k) or (\alpha \pm mb^k)$.
if $m = ka$ is a multiple of a, then the element $(\alpha \mp kb^{k+1})$ is added to the CC.
The maximum number of elements having pattern $(\alpha \mp kb^{k+1}) \in$ [LB,UB] is $\frac{R}{|b|^k}$. Hence Proved.

The upper bound No. of CCs of length 1 = R. If the selected range does not have any solution to LDe then all iterations are independent.

2.6 Computing Seeds of Single LDE

The set of dependent CCs are solutions of LDE. Slope of LDE is significant for computation of solutions. Type of solutions based on slope are:
$Slope = 1$ for LDE $x - y = c$, data dependence is constant, c CCs are formed.
$Slope = -1$ for LDE $x + y = c$, VD^3 case, all CCs have length 2.
$0 < |Slope| < 1$ VD^3 case, CCs vary in length and number.

The Algorithm for computing Seeds:

1. S (= set of seeds) = { }; R = [L,U];
2. $slope = -a/b$; $\alpha = c/(a+b)$
3. if ((α is integer) AND ($L \leq \alpha \leq U$)) { Add α to S. }

```
4. if( slope == 1 ) S+=[L,L+c-1]
5. else if ( slope == -1 )
6. { if ($L < \alpha < R$) S+=[L,$\alpha$]+[$\alpha$+L,R]
7. else if($R < \alpha$——$\alpha < L$) S+=[L,R] }
8. else { Compute $p_{max}$, such that $R/(|b|^{P_{max+1}}) < 1 \leq R/(|b|^{P_{max}})$.
9. $L_{max}(= Lengthoflongest CC) = P_{max} + 1$;
10. for l = 2 to $Lmax$ {
11. for $n \leq \frac{R}{b^{l-1}}$ { m = 1;
12. if (m is a not a multiple of a or b )
13. { if L < $\alpha - m.b^i$ add it to S
14. else add the least element > L to S from same CC}
15. { if L < $\alpha + m.a^i$ add it to S
16. else add the least element > L to S from same CC}
17. $m = m + 1$. } } }
```

Consider the example $2x - 3y = 5$ and R = [-8,20], shown in figure 2.1:
α = -5, -5 is a seed.
$p_{max} = 3$, $l_{max} = 4$
-5 + 1*(-3) = -8 and -5 + 1*2 = -3 are seeds.
similarly other seeds are -5, -1, 3, 5, 9, 11.

2.7 Non-Integer α

IF c is not a multiple of $a+b$, α is a not an integer but a real number. Using parametric solution to LDE [?] CCs and Seeds can be computed.

- Find a solution of LDE $\{\beta, \gamma\}$ such that $|\beta - \alpha| < |b|$ and $|\gamma - \alpha| < a$. Such a solution exists since a pair of consecutive integer solutions of LDE have x and y values -b and a iterations apart respectively [?] and α is a floating point number in between such a pair of solutions.
- Other solutions of LDE are of the form $\beta \pm mb, \gamma \mp ma$, where m is an integer(Using Theorem 1).
- Longer CCs are formed when $\{\beta \pm m_1 b = \gamma \mp m_2 a\}$. They are of the form $\{\beta \pm m_2 b, \gamma \mp m_2 a = \beta \pm m_1 b, \gamma \mp m_1 a\}$.
- The upper bounds on number and length of CCs depend on b and R and not on c. Hence upper bounds on number and length of CCs do not change if α is not an integer.

An example of the stated formulation is presented here.

- Consider an LDE $3x - 5y = 3$. Here $\alpha = -3/2 = -1.5$ is not an integer.
- The solutions (1,0) and (-4,-3) on either side of α are 2 such solutions.
- In the given example, form of other solutions are $\{1 + 5m, 0 + 3m\}$ to right and $\{-4 - 5m, -3 - 3m\}$ to left of α.
- For this example longer CC is formed for $\{1 + 2 * 5, 0 + 2 * 3 = 1 + 1 * 5, 0 + 1 * 3\}$ i.e. $\{11, 6, 3\}$

2.8 General form of Solutions of LDE

The ratio of $\gamma - \alpha : \beta - \alpha = -a : b$ (using Slope-Point form of equation of a line).

Theorem 2: The general form of a solution $\{\beta, \gamma\}$ of an LDE $ax + by = c$ is:
$\{(c - kb)/(a + b), (c + ka)/(a + b)\}$, where k is an integer.

Proof: Assume $\alpha = c/(a + b)$ is not an integer.
The general form of a solution is $\{\beta, \gamma\}$ where, β and γ are integers.
Let z_1 and z_2 be 2 integers such that $\beta = (c + z_1)/(a + b)$ and $\gamma = (c + z_2)/(a + b)$
Then:
$\therefore \gamma - \alpha = z_2/(a + b)$ and $\beta - \alpha = z_1/(a + b)$
$\therefore$ using slope point form, $z_2/z_1 = -a/b$
$\therefore z_2/a = -z_1/b$
Let k be an integer such that $z_2/a = k$ then $z_1/b = -k$
$\therefore \{(c - kb)/(a + b), (c + ka)/(a + b)\}$ is a solution to the LDE for any $k \in Z$
Therefore the general form of a solution $\{\beta, \gamma\}$ of an LDE is:
$(c - kb)/(a + b), (c + ka)/(a + b)$.
When α is an integer $k = 0$. Hence proved.

Hence the generalized condition for formation of longer CCs is:
$(c \mp k_2 b)/(a + b) \mp m_2 b = (c \pm k_1 a)/(a + b) \pm m_1 a$, which is possible if
k_2 and m_2 are multiples of a.
k_1 and m_1 are multiples of b.
Using Theorem 1 and 2, the general form of seeds of a CC is $(c + ka^x b^y)/(a + b) + ma^i b^j$, where $i + j$ = length of CC, i,j,x and y are positive integers and k and m are integers.

Let $\{\beta, \gamma\}$ be a solution for $ax + by = c$
$\therefore a\beta + b\gamma = c$
$\therefore a(\beta \pm d) + b(\gamma \pm d) = a\beta + b\gamma \pm d(a + b) = c \pm d(a + b)$
$\therefore \{\beta \pm d, \gamma \pm d\}$ is a solution for $ax + by = c \pm d(a + b)$.

For example (1,0) is a solution for $3x - y = 3$
a = 3, b = -1 , a+b = 2
(2,1) = (1+1,0+1) is a solution for $3x - y = 3 + 2 = 5$
(0,-1) = (1-1, 0-1) is a solution for $3x - y = 3 - 2 = 1$
Hence solutions of LDEs with other values of c can be generated using solutions for $0 \leq c < a+b$.

2.9 Test for Independent Iteration

Using the general form of solutions $\beta \pm mb, \gamma \mp ma$, the algorithm for testing an independent iteration t is given in this section.

Algorithm for test of Independent Iteration

```
1. Let I = {} set of independent iterations
2. Let D = {} set of dependent iterations
3. Find particular solution of LDE ( $\beta$ , $\gamma$ )
4. for i = LB to UB do
5. if ( (i - $\beta$) is divisible by b ) add i to D
6. else if ( (i - $\gamma$) is divisible by a ) add i to D
7. else add i to I
```

For iteration values 4 and 6 and LDE $2x - 3y = 5$, $\{-5, -5\}$ is a particular solution.
(4 -(-5)) = 9 is divisible by b (= -3) so 4 is a dependent iteration.
(6 -(-5)) = 11 is not divisible by both a (= 2) and b (= -3) hence 6 is an independent iteration.

The sets I and D will contain independent and dependent iterations respectively. CCs of dependent iterations in D can be formed using the method given in previous sections of this report. Schedules for dependent iterations with maximum 1 thread per CC can be generated. The independent iterations from I can be scheduled in a way to balance the load on scheduling threads.

2.10 Heuristic based Splitting

Coefficients of LDE can either have same sign or different signs. If coefficients of LDE have same sign then large number of dependence edges cross iteration $\alpha = c/(a + b)$. For example consider the following loop accessing array g. Its corresponding LDE is: x + 2y = 3.

```
for(i = -10 ; i < 10 ; i++)
{ S1 g[i] = .........;
  S2 ......... = g[3-2i];}
```

The solution of LDE for the range R = [-8,10], in (sink,source) form is: { (-7,5), (-5,4), (-3,3), (-1,2), (**1**,**1**), (3,0), (5,-1), (7,-2), (9,-3) }. The dependence is as shown in Figure 2.2. The partition P = { {**-7**,5,-1,2}, {**-5**,4}, {**-3**,3,0,9}, {**1**}, {**-2**,7} }.

The above example highlights the importance of loop independent iteration *alpha* for splitting. Following are some observations about α:

- α has RAW and WAR data dependence on either side of it.

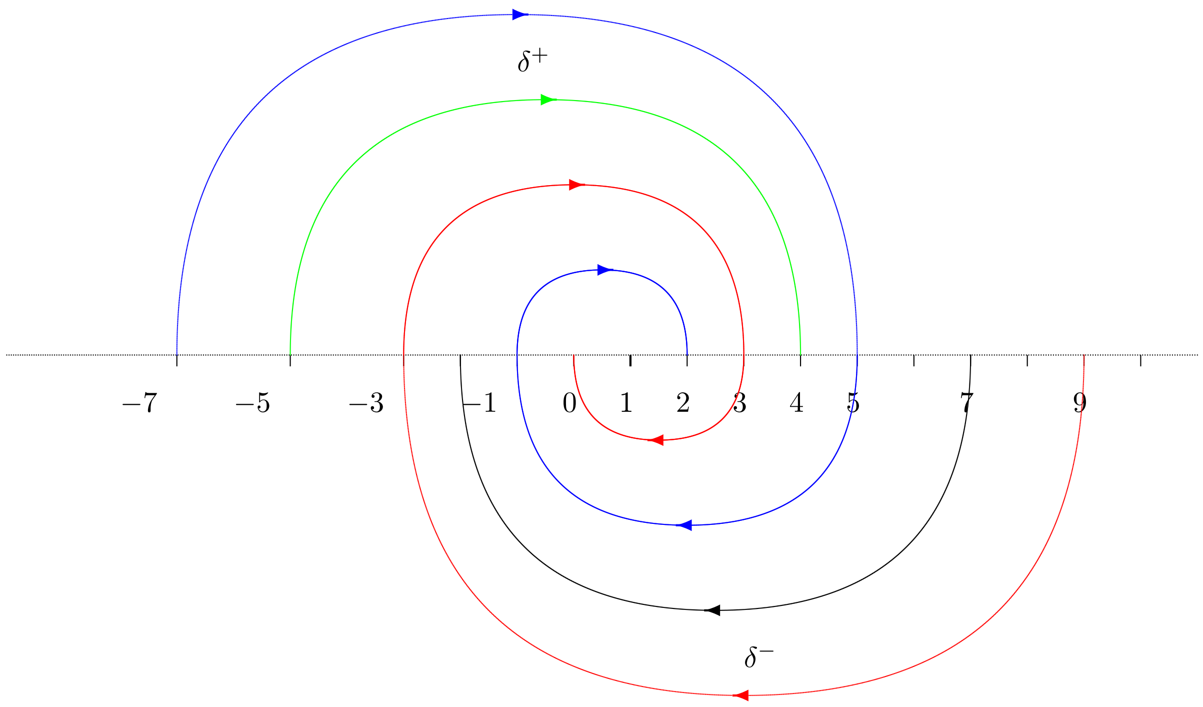

Figure 2.2: CCs of x+2y=3

- The data dependence distance between a pair of iterations increases as the distance between α and the iterations increases.
- If α is in the range then maximum number of dependence edges cross it.

For such cases, if iteration $\alpha \in$ [LB, UB], splitting the Iteration range [LB,UB] into 2 ranges i.e. [LB,α] and [α+1,UB], largely improves the exploitable parallelism.

The steps in splitting are:

1. LDE representing the underlined dependence relation: $x + 2y = 3$.
2. Partition without splitting P = { {-9, 6}, {-7, -1, 2, 5}, {-5, 4}, {-3, 0, 3, 9}, {-2, 7}, {1} }.
3. Independent iterations I = {-10, -8, -6, -4, -2, 8, 10}.
4. $\alpha = 1$.
5. splitting range R = [-10 , 10] to R_1 = [-10 , 1] and R_2 = [2 , 10].
6. Partitions after splitting P_1 = { {1} } and P_2 = { {} } as CC {-3, 0, 3, 9} is formed by (-3,3), (0,3) and (-3,9) neither of the source sink pairs are in same slice.
7. Independent iterations of 1st slice I_1 = {-10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0}.
8. Independent iterations of 2nd slice I_2 = {2, 3, 4, 5, 6, 7, 8, 9, 10}.

This example shows remarkable improvement in exploitable parallelism.

Splitting the range [LB,UB] into 2 ranges i.e. [LB,α] and [α+1,UB] largely improves the exploitable parallelism in case of 2 variable LDEs. In case of multiple, 2 variable LDEs heuristic based approach for splitting iteration space efficiently is used. If independent iterations αs of multiple 2 variable LDEs are in the range $[L_\alpha, U_\alpha]$, a suitable split point α_h will be $(L_\alpha + U_\alpha)/2$. The results of such a splitting are presented in chapter 3.

Capturing dependence in case of loops with VD^3 for a two variable is presented in this Chapter. Our approach uses a two variable LDE and its parametric solutions to analyze and mathematically formulate Precise Dependence, the inherent dependence in the loops. Theoretical methods to compute precise dependence are presented. The result is a partition of the iteration space, a set of parallel CCs. We computed bounds on the number and size of CCs. The partition is represented in a reduced form by a set of seeds. Next Chapters extend this approach for multiple loops with and without multiple LDEs.

Chapter 3

Results and Implementation

3.1 Experimental Results

The analysis and algorithms developed and presented in this paper show how partitioning of iteration space can be done in case of VD^3 and multiple loops with multiple LDEs. We have considered 2 variable LDEs with separable loops. In the presence of multiple LDEs and the ensuing interactions between their individual partitions the generation of the overall partition of the system is difficult. While our formulation successfully captures the overall partition, the experimentation explores the extent of existing parallelism.

The experiments were parametrized by: i) Loop range ii) number of LDEs iii) Randomized coefficients for VD_3 and iv) number of experiments carried out. The loop range was varied from [-2 to 2,..., -2^{15} to 2^{15}], [1..90] LDEs and up to 100 experiments. For each experiment the partition was evaluated in terms of minimum, maximum and average number of CCs, which are measures of exploitable parallelism.

- Table 3.1 shows that the extent of parallelism on the average is atleast 95% across various ranges. Thus, VD^3 for single LDE has significant exploitable parallelism even for large loop ranges.
- Table 3.2 shows average parallelism when the number of LDEs are increased up to 90. The number of parallel components reduce considerably when the number of LDEs increase. This is a consequence of interaction between the partitions of the individual LDEs. Figure 3.1 shows the range of overall exploitable parallelism for 1,30 and 90 LDEs. The table 3.2 shows average behaviour and graph shows maximum parallelism. It can be seen that as the as the number of LDEs increase the maximum parallelism decreases. There is not enough difference in the minimum parallelism is
- Results show that exploitable parallelism decreases for increasing number of loops. It was observed that a large number of dependences cross loop independent iterations (αs) in solutions of LDEs. We computed a representative iteration αr using heuristic based on set of loop independent iterations (αs). The range was split into [LB,αr] and [αr+1,UB]. The partition for all the LDEs reported in table 3.2 have been recreated using the heuristic. This is reported in table 3.3 it shows a significant improvement in exploitable parallelism if the range is split into [LB , α_h] and [$\alpha_h + 1$, UB]. The improvement ratio increases to approximately 20:1 as the number of LDEs and range increase. Figure 3.2 shows significant improvement in maximum parallelism for 1,30 and 90 LDEs after splitting the range into 2 ranges. Comparing Figure 3.2 and Figure 3.1 it can be observed that the effective

Range	Parallels	CCs	Independents
5	5	0	5
9	9	0	9
17	16.97	0.03	16.94
33	32.93	0.07	32.86
65	64.81	0.18	64.63
129	128.44	0.54	127.9
257	255.27	1.67	253.6
513	508.42	4.35	504.07
1025	1012.16	12.32	999.84
2049	2016.82	30.92	1985.9
4097	4021.72	72.44	3949.28
8193	8026.04	161.06	7864.98
16385	16036.87	335.95	15700.92
32769	32058.47	685.59	31372.88
65537	64101.76	1384.98	62716.78

Table 3.1: Average exploitable parallelism for single LDE

Range	2LDE	5LDEs	30LDEs	90LDEs
5	4.97	4.93	4.61	3.99
9	8.92	8.89	8.16	6.78
17	16.82	16.69	15.11	11.74
33	32.59	32.39	28.57	20.66
65	64.15	63.28	54.06	34.68
129	127.11	124.20	99.11	51.44
257	252.79	244.76	176.12	68.40
513	501.27	481.27	308.30	87.65
1025	993.81	945.15	525.85	114.74
2049	1972.04	1852.66	897.32	150.58
4097	3920.22	3646.23	1557.49	211.95
8193	7810.89	7224.94	2853.74	346.77
16385	15599.41	14388.60	5488.09	628.31
32769	31176.54	28714.82	10722.84	1159.05
65537	62330.68	51724.69	21179.73	2199.60

Table 3.2: Average exploitable parallelism for varying No. of LDEs

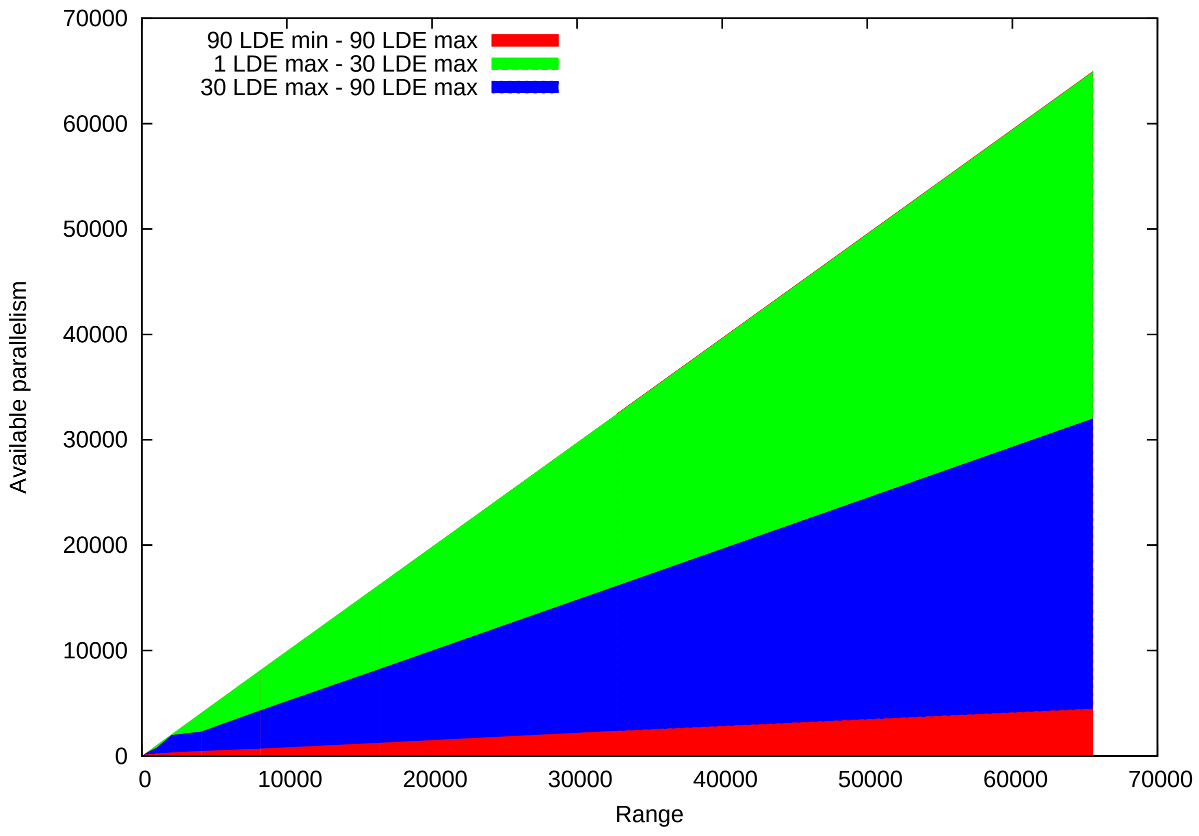

Figure 3.1: Overall Exploitable parallelism

maximum parallelism shows considerable improvement. To summarize the heuristic used by us to split the iteration ranges seems to profitable.

	30LDEs		90LDEs	
Range	Avg	split-Avg	Avg	split-Avg
5	4.61	4.68	3.99	4.15
9	8.16	8.49	6.78	6.93
17	15.11	15.57	11.74	12.23
33	28.57	29.29	20.66	21.44
65	54.06	56.10	34.68	37.73
129	99.11	108.25	51.44	70.25
257	176.12	217.2	68.40	144.19
513	308.30	439.07	87.65	318.07
1025	525.85	880.85	114.74	660.92
2049	897.32	1754.99	150.58	1330.00
4097	1557.49	3514.13	211.95	2651.60
8193	2853.74	7065.49	346.77	5410.43
16385	5488.09	14264.31	628.31	11192.73
32769	10722.84	28702.30	1159.05	22849.37
65537	21179.73	57578.68	2199.60	46151.75

Table 3.3: Improved Average exploitable parallelism after range splitting

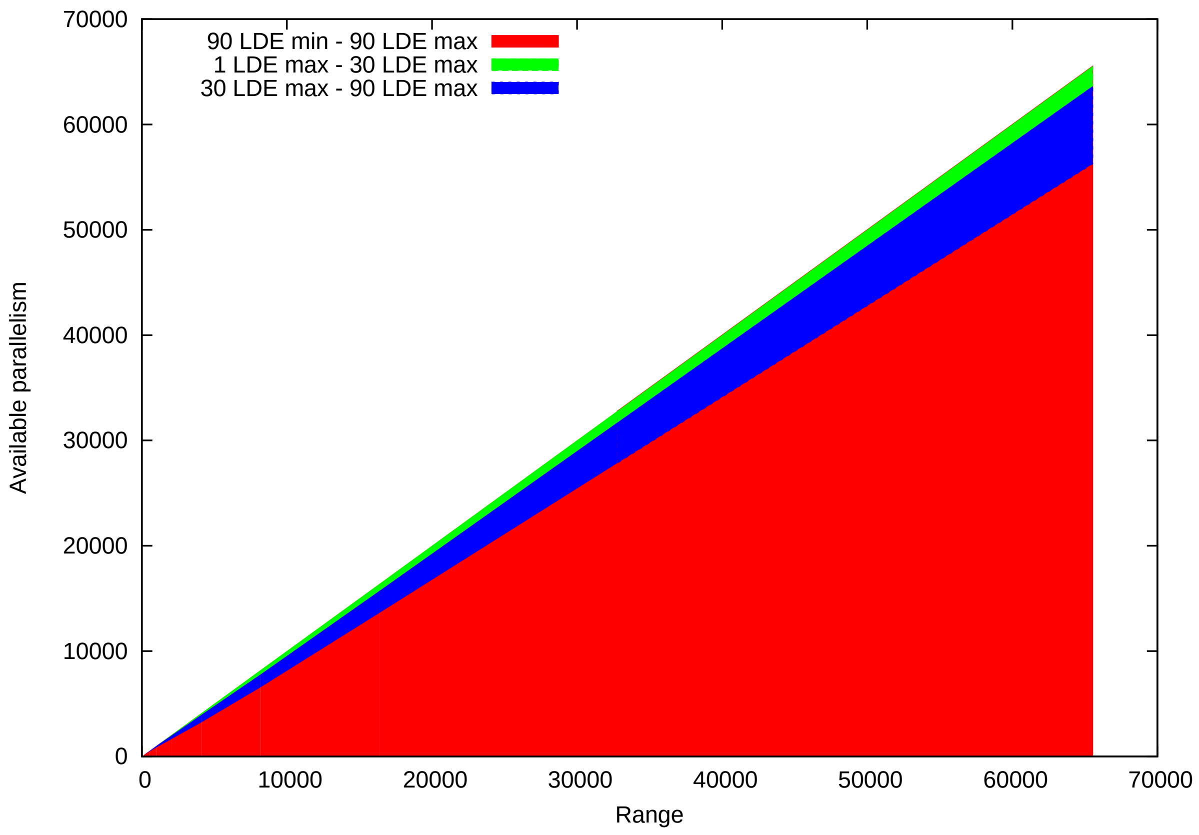

Figure 3.2: Exploitable parallelism after splitting

Chapter 4

Conclusion

Capturing dependence in case of loops with Variable Distance Data Dependence(VD^3) is presented in this report. Given an Iteration Space(IS) and a set of LDEs, our notion of Precise Dependence(PD) denotes the collection of all solutions of all LDEs in it. Iterations which do not appear in PD are independent iterations. PD captures all the inherent parallelism present at iteration level granularity. Computing PD in the most general case is NP-complete. Dependences in most of the array accesses found in practice follow a pattern and hence are track-able. These patterns of dependences make PD amenable to our approach. Our handling of exploitable parallelism in VD^3 and multiple loops expands the domain of applications that are parallelizable today. Compile time analysis of PD opens opportunity for automatic generation of parallel threads suited for multi-core configuration. Even heavily connected dependence graphs show considerable parallelism which can be exploited by a well designed scheduling algorithm.

4.1 Summary of Contributions

Variable Distance Data Dependence has an irregular and complex structure of underlined dependence making it difficult to analyze and compute. Our approach uses a two variable LDE and its parametric solutions to analyze and mathematically formulate PD, the inherent dependence in the loops. Theoretical methods to compute PD are presented in this report.

During the first Annual Progress Seminar the work presented was based on parametric solutions of source-sink pairs computed for a given iteration space[?]. This result was examined in an attempt to study the possible existence of patterns of dependence. This led to the discovery of Connected Components(CCs), a set of all inter-dependent iterations. These CCs are non trivial because of the VD^3 that they capture. These CCs have been used to partition the iteration space. The bounds on number and length of CCs were computed for a given iteration space. Further probing showed that a CC can be represented by a single iteration belonging to it called Seed. As a result, the dependence structure in an iteration space can be represented in a compressed form by a set of Seeds (1 per CC). It was observed that these partitions made such loops amenable to parallelism.

In the second Annual Progress Seminar, generalized partitions of 2 variable LDEs having integer or non-integer values of loop independent iteration, α was reported. Test for independent iterations was presented. Heuristic based splitting was presented which noticeably enhances the exploitable parallelism in case of 2 variable LDEs. Our findings had shown that it was possible to reduce Multi Variable LDE to a set of 2 Variable LDEs. We also found that solving these 2

Variable LDEs and combining the solutions thus obtained, generated solution of Multi Variable LDE using cross product. This concept was extended to a system of LDEs. The solutions of individual Multi Variable LDEs, from a system of LDEs were obtained and merged using pair wise intersection between CCs called Fmerge in a non trivial way.

We randomly created multiple LDEs in large number for our experimentation. Results showed existence of reasonable parallelism which reduced as the number of LDEs increased. To improve results we had formed an alternate partition using a heuristic. This heuristic showed non trivial improvement in parallelism.

Our work this year showed that cross product based merging generates spurious dependences in some cases. This was improved upon and a method based on cross product over iteration vectors from CCs taken pairwise was developed. This honored inherent dependences and eliminated spurious dependences. We further realized that depending on the context of array accesses, solutions of multi-variable LDEs need to be merged in different ways to obtain solutions of system of LDEs. The 2 ways of merging reported are: a simple intersection of solutions and $\uplus$, an overlap based modified union of solutions. Our approach handles separable, partially separable and non-separable subscript expressions. The solution thus obtained is precise.

Contributions this year are:

- The systematic construction of LDEs based on the context of array accesses is introduced.
- The merging of solutions of 2 variable LDEs using cross product was found to generate spurious dependences. This was reworked and a method of taking pair wise cross product is introduced. The results of this method were rigorously tested and compared with exhaustive solutions for correctness of precise dependence.
- Fmerge, merging method presented in last APS is replaced by operator $\uplus$. Mathematical formulations of $\uplus$ for computation of overlap between CCs is a presented in this report. Use of $\uplus$ strengthens the approach.
- While computing solution of system of LDEs the context of array accesses was found be the deciding factor in selecting the merging method. LDEs representing different subscript positions of pair of accesses are merged by intersection. LDEs representing reference pairs of array access are merged using $\uplus$.
- Our approach is presented in algorithmic way.
- With respect to array subscripts, the contextual modification in our approach seamlessly handles separable, partially separable and non-separable array accesses.
- Due to changes in the approach, the results of comparison with examples presented in literature were reworked. These results show comparable performance of PDP with literature.

Our contribution is in identifying patterns in solutions of Linear Diophantine Equation(LDE). We determine a minimal subset of solutions from which complete solution can be generated or 2 variable LDE. We further extend it to compose solutions Multi Variable LDEs and system of LDEs by merging constituent solutions in non trivial way. The notion of Precise Dependence used by us extracts all the parallelism that is present in loops with non-uniform distances. The parallelism thus obtained is comparable to the results found in literature. Thus Precise dependence, its resultant partition and exploitable parallelism has been obtained array accesses in rectangular iteration space.

The next sections presents directions for further work.

4.2 Directions for Further Work:

The following points indicate directions of further work to be done (not in any specific order) including some issue that need to be addressed for generating solutions in a more generalized way.

1. For a single 2 variable LDE splitting based on loop independent iteration α(section 2.10) shows noticeable improvement in parallelization. As the number of LDEs used to generate precise dependence increase in number and number of variables, finding a suitable iteration for splitting appears to be a complex process. Finding a suitable splitting method which may or may not use heuristic needs to be studied. Effect of splitting on parallelization of needs to be addressed.

2. Comparing the partitions generated by PDP with parallelization by state of art tools like Pluto.

3. Extending this approach for having trapezoidal bounds. An example code is given below.

```
for( j = 0;j < 100 ; j++)
for( i = j + 10; i < j + 100 ; i++)
{
a[2i + 1][2j + 4] = i;
b[i] = a[3i + 6][j] + i;
}
```

4. Theoretical comparison of PDP with approaches presented in literature.

5. Generating efficient schedules using precise dependence analysis. Generating schedules is in itself a larger research problem. We intend to evaluate the practicality of using partitions computed by PDP for generation of usable schedules.

6. Experimentally evaluating PDP for Benchmarks.

7. Prototype implementation using suitable architecture.

Appendix A

Result of data dependence pass of gcc on Loops

This Chapter presents the result of data dependence pass of gcc on loops with constant and variable distance data dependence.

A.1 Constant distance data dependence

Given below is the code and corresponding result of data dependence pass of gcc on loop with constant distance data dependence.

Code

```
#include<stdio.h>
int a[200],b[200];
void main()
{
int i,j;
for( i = 0;i < 150 ; i++)
{
a[2*i + 1] = i;
b[i] = a[3*i + 6] + i;
}
}
```

Output of data dependence pass of gcc

```
;; Function main (main)


Pass statistics:
----------------

Creating dr for a[i_3]
analyze_innermost: (analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = &a)
```

```
(get_scalar_evolution
  (scalar = &a)
  (scalar_evolution = ))
)
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = (unsigned int) i_3 * 4)
(get_scalar_evolution
  (scalar = (unsigned int) i_3 * 4)
  (scalar_evolution = ))
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = (unsigned int) i_3)
(get_scalar_evolution
  (scalar = (unsigned int) i_3)
  (scalar_evolution = ))
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = i_3)
(get_scalar_evolution
  (scalar = i_3)
  (scalar_evolution = ))
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = i_13)
(get_scalar_evolution
  (scalar = i_13)
  (scalar_evolution = ))
(analyze_initial_condition
  (loop_phi_node =
i_13 = PHI <i_3(4), 0(2)>
)
  (init_cond = 0))
(analyze_evolution_in_loop
  (loop_phi_node = i_13 = PHI <i_3(4), 0(2)>
)
(add_to_evolution
  (loop_nb = 1)
  (chrec_before = 0)
  (to_add = 1)
  (res = {0, +, 1}_1))
  (evolution_function = {0, +, 1}_1))
(set_scalar_evolution
  instantiated_below = 2
  (scalar = i_13)
  (scalar_evolution = {0, +, 1}_1))
)
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = 1)
(get_scalar_evolution
  (scalar = 1)
  (scalar_evolution = 1))
)
(set_scalar_evolution
  instantiated_below = 2
  (scalar = i_3)
  (scalar_evolution = {1, +, 1}_1))
)
)
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = 4)
(get_scalar_evolution
  (scalar = 4)
  (scalar_evolution = 4))
)
)
success.
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = i_3)
(get_scalar_evolution
  (scalar = i_3)
  (scalar_evolution = {1, +, 1}_1))
(set_scalar_evolution
  instantiated_below = 2
  (scalar = i_3)
  (scalar_evolution = {1, +, 1}_1))
)
(instantiate_scev
  (instantiate_below = 2)
  (evolution_loop = 1)
  (chrec = {1, +, 1}_1)
  (res = {1, +, 1}_1))
base_address: &a
offset from base address: 0
constant offset from base address: 4
step: 4
aligned to: 128
base_object: a[0]
Creating dr for a[D.2754_4]
analyze_innermost: (analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = &a)
(get_scalar_evolution
  (scalar = &a)
  (scalar_evolution = ))
```

```
)
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = (unsigned int) D.2754_4 * 4)
(get_scalar_evolution
  (scalar = (unsigned int) D.2754_4 * 4)
  (scalar_evolution = ))
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = (unsigned int) D.2754_4)
(get_scalar_evolution
  (scalar = (unsigned int) D.2754_4)
  (scalar_evolution = ))
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = D.2754_4)
(get_scalar_evolution
  (scalar = D.2754_4)
  (scalar_evolution = ))
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = i_13)
(get_scalar_evolution
  (scalar = i_13)
  (scalar_evolution = {0, +, 1}_1))
(set_scalar_evolution
  instantiated_below = 2
  (scalar = i_13)
  (scalar_evolution = {0, +, 1}_1))
)
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = 6)
(get_scalar_evolution
  (scalar = 6)
  (scalar_evolution = 6))
)
(set_scalar_evolution
  instantiated_below = 2
  (scalar = D.2754_4)
  (scalar_evolution = {6, +, 1}_1))
)
)
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = 4)
(get_scalar_evolution
  (scalar = 4)
  (scalar_evolution = 4))
```

```
)
)
success.
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = D.2754_4)
(get_scalar_evolution
  (scalar = D.2754_4)
  (scalar_evolution = {6, +, 1}_1))
(set_scalar_evolution
  instantiated_below = 2
  (scalar = D.2754_4)
  (scalar_evolution = {6, +, 1}_1))
)
(instantiate_scev
  (instantiate_below = 2)
  (evolution_loop = 1)
  (chrec = {6, +, 1}_1)
  (res = {6, +, 1}_1))
base_address: &a
offset from base address: 0
constant offset from base address: 24
step: 4
aligned to: 128
base_object: a[0]
Creating dr for b[i_13]
analyze_innermost: (analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = &b)
(get_scalar_evolution
  (scalar = &b)
  (scalar_evolution = ))
)
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = (unsigned int) i_13 * 4)
(get_scalar_evolution
  (scalar = (unsigned int) i_13 * 4)
  (scalar_evolution = ))
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = (unsigned int) i_13)
(get_scalar_evolution
  (scalar = (unsigned int) i_13)
  (scalar_evolution = ))
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = i_13)
(get_scalar_evolution
  (scalar = i_13)
  (scalar_evolution = {0, +, 1}_1))
(set_scalar_evolution
  instantiated_below = 2
  (scalar = i_13)
  (scalar_evolution = {0, +, 1}_1))
)
)
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = 4)
(get_scalar_evolution
  (scalar = 4)
  (scalar_evolution = 4))
)
)
success.
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = i_13)
(get_scalar_evolution
  (scalar = i_13)
  (scalar_evolution = {0, +, 1}_1))
(set_scalar_evolution
  instantiated_below = 2
  (scalar = i_13)
  (scalar_evolution = {0, +, 1}_1))
)
(instantiate_scev
  (instantiate_below = 2)
  (evolution_loop = 1)
  (chrec = {0, +, 1}_1)
  (res = {0, +, 1}_1))
base_address: &b
offset from base address: 0
constant offset from base address: 0
step: 4
aligned to: 128
base_object: b[0]
(compute_affine_dependence
  (stmt_a =
a[i_3] = i_13;
)
  (stmt_b =
D.2755_5 = a[D.2754_4];
)
(subscript_dependence_tester
(analyze_overlapping_iterations
  (chrec_a = {1, +, 1}_1)
  (chrec_b = {6, +, 1}_1)
(analyze_siv_subscript
(analyze_subscript_affine_affine
  (overlaps_a = [5 + 1 * x_1]
)
  (overlaps_b = [0 + 1 * x_1]
)
)
)
  (overlap_iterations_a = [5 + 1 * x_1]
)
  (overlap_iterations_b = [0 + 1 * x_1]
)
)
(build_classic_dist_vector
  dist_vector = (  5
  )
)
)


Banerjee Analyzer
(Data Dep:
#(Data Ref:
#  stmt: a[i_3] = i_13;
#  ref: a[i_3];
#  base_object: a[0];
#  Access function 0: {1, +, 1}_1
#)
#(Data Ref:
#  stmt: D.2755_5 = a[D.2754_4];
#  ref: a[D.2754_4];
#  base_object: a[0];
#  Access function 0: {6, +, 1}_1
#)
  access_fn_A: {1, +, 1}_1
  access_fn_B: {6, +, 1}_1

 (subscript
  iterations_that_access_an_element_twice_in_A: [5 + 1 * x_1]
  last_conflict: 145
  iterations_that_access_an_element_twice_in_B: [0 + 1 * x_1]
  last_conflict: 145
  (Subscript distance: 5
  )
 )
  inner loop index: 0
  loop nest: (1 )
  distance_vector:   5
  direction_vector:     +
)

omega_solve_eq (3, 0)
variables = protected (a), b
5 + a = 0
Done with EQ
!0 <= b
!0 <= a + b
!b <= 150
!a + b <= 150
!0 <= 150 + a
!a <= 150
!0 <= a
Done with GEQ

----
eliminating variable a
substituting using a := -5
variables = protected (a), b
!10 <= 0
!0 <= 160
!0 <= 140
!b <= 160
!b <= 150
!10 <= b
!0 <= b
---

variables = protected (a), b
Done with EQ
!0 <= b
!10 <= b
!b <= 150
!b <= 160
!0 <= 140
!0 <= 160
!10 <= 0
Done with GEQ
===


omega_solve_geq (3,0):
variables = b
Done with EQ
!0 <= b
!10 <= b
!b <= 150
!b <= 160
```

```
!0 <= 140
!0 <= 160
!10 <= 0
Done with GEQ
a := -5


equations have no solution

omega_solve_eq (3, 0)
variables = protected (a), b
a = 5
Done with EQ
!0 <= b
!0 <= a + b
!b <= 150
!a + b <= 150
!0 <= 150 + a
!a <= 150
!0 <= a
Done with GEQ

----
eliminating variable a
substituting using a := 5
variables = protected (a), b
!0 <= 10
!0 <= 140
!0 <= 160
!b <= 140
!b <= 150
!0 <= 10 + b
!0 <= b
---

variables = protected (a), b
Done with EQ
!0 <= b
!0 <= 10 + b
!b <= 150
!b <= 140
!0 <= 160
!0 <= 140
!0 <= 10
Done with GEQ
===


omega_solve_geq (3,0):
```

```
variables = b
Done with EQ
!0 <= b
!0 <= 10 + b
!b <= 150
!b <= 140
!0 <= 160
!0 <= 140
!0 <= 10
Done with GEQ
a := 5

upper bound = 140
lower bound = 0
Doing chain reaction unprotection
variables =
Done with EQ
Done with GEQ
a := 5
After chain reactions
variables =
Done with EQ
Done with GEQ
a := 5
-------------------------------------------
problem reduced:
variables =
Done with EQ
Done with GEQ
a := 5
-------------------------------------------
Omega Analyzer
(Data Dep:
#(Data Ref:
#  stmt: a[i_3] = i_13;
#  ref: a[i_3];
#  base_object: a[0];
#  Access function 0: {1, +, 1}_1
#)
#(Data Ref:
#  stmt: D.2755_5 = a[D.2754_4];
#  ref: a[D.2754_4];
#  base_object: a[0];
#  Access function 0: {6, +, 1}_1
#)
  access_fn_A: {1, +, 1}_1
  access_fn_B: {6, +, 1}_1

 (subscript
  iterations_that_access_an_element_twice_in_A: [5 + 1 * x_1]
  last_conflict: 145
  iterations_that_access_an_element_twice_in_B: [0 + 1 * x_1]
  last_conflict: 145
  (Subscript distance: 5
  )
 )
  inner loop index: 0
  loop nest: (1 )
  distance_vector:   5
  direction_vector:     +
)
)
(compute_affine_dependence
  (stmt_a =
a[i_3] = i_13;
)
  (stmt_b =
b[i_13] = D.2755_5;
)
)
(compute_affine_dependence
  (stmt_a =
D.2755_5 = a[D.2754_4];
)
  (stmt_b =
b[i_13] = D.2755_5;
)
)
Dependence tester statistics:
Number of dependence tests: 1
Number of dependence tests classified dependent: 1
Number of dependence tests classified independent: 0
Number of undetermined dependence tests: 0
Number of subscript tests: 1
Number of undetermined subscript tests: 0
Number of same subscript function: 0
Number of ziv tests: 0
Number of ziv tests returning dependent: 0
Number of ziv tests returning independent: 0
Number of ziv tests unimplemented: 0
Number of siv tests: 1
Number of siv tests returning dependent: 1
Number of siv tests returning independent: 0
Number of siv tests unimplemented: 0
Number of miv tests: 0
Number of miv tests returning dependent: 0
Number of miv tests returning independent: 0
Number of miv tests unimplemented: 0
```

```
(Data Dep:
#(Data Ref:
#  stmt: a[i_3] = i_13;
#  ref: a[i_3];
#  base_object: a[0];
#  Access function 0: {1, +, 1}_1
#)
#(Data Ref:
#  stmt: D.2755_5 = a[D.2754_4];
#  ref: a[D.2754_4];
#  base_object: a[0];
#  Access function 0: {6, +, 1}_1
#)
  access_fn_A: {1, +, 1}_1
  access_fn_B: {6, +, 1}_1

 (subscript
  iterations_that_access_an_element_twice_in_A: [5 + 1 * x_1]
  last_conflict: 145
  iterations_that_access_an_element_twice_in_B: [0 + 1 * x_1]
  last_conflict: 145
  (Subscript distance: 5
  )
 )
  inner loop index: 0
  loop nest: (1 )
  distance_vector:   5
  direction_vector:     +
)
(Data Dep:
#(Data Ref:
#  stmt: a[i_3] = i_13;
#  ref: a[i_3];
#  base_object: a[0];
#  Access function 0: {1, +, 1}_1
#)
#(Data Ref:
#  stmt: b[i_13] = D.2755_5;
#  ref: b[i_13];
#  base_object: b[0];
#  Access function 0: {0, +, 1}_1
#)
    (no dependence)
)
(Data Dep:
#(Data Ref:
#  stmt: D.2755_5 = a[D.2754_4];
#  ref: a[D.2754_4];
#  base_object: a[0];
#  Access function 0: {6, +, 1}_1
#)
#(Data Ref:
#  stmt: b[i_13] = D.2755_5;
#  ref: b[i_13];
#  base_object: b[0];
#  Access function 0: {0, +, 1}_1
#)
    (no dependence)
)


DISTANCE_V (  5
)
DIRECTION_V (    +
)


(classify_chrec {1, +, 1}_1
  affine_univariate
)
(classify_chrec {6, +, 1}_1
  affine_univariate
)
(classify_chrec {0, +, 1}_1
  affine_univariate
)

(
-----------------------------------------
3 affine univariate chrecs
0 affine multivariate chrecs
0 degree greater than 2 polynomials
0 chrec_dont_know chrecs
-----------------------------------------
3 total chrecs
0 with undetermined coefficients
-----------------------------------------
3 chrecs in the scev database
8 sets in the scev database
22 gets in the scev database
-----------------------------------------
)


Pass statistics:
----------------

main ()
```

```
{
  intD.0 pretmp.2D.2759;
  intD.0 iD.1875;
  intD.0 D.2755;
  intD.0 D.2754;

  # BLOCK 2 freq:100
  # PRED: ENTRY [100.0%]  (fallthru,exec)
  # SUCC: 3 [100.0%]  (fallthru,exec)

  # BLOCK 3 freq:9900
  # PRED: 4 [100.0%]  (fallthru,dfs_back,exec) 2 [100.0%]  (fallthru,exec)
  # iD.1875_13 = PHI <iD.1875_3(4), 0(2)>
  # .MEMD.2756_14 = PHI <.MEMD.2756_10(4), .MEMD.2756_8(D)(2)>
  iD.1875_3 = iD.1875_13 + 1;
  # .MEMD.2756_9 = VDEF <.MEMD.2756_14>
  aD.1871[iD.1875_3] = iD.1875_13;
  D.2754_4 = iD.1875_13 + 6;
  # VUSE <.MEMD.2756_9>
  D.2755_5 = aD.1871[D.2754_4];
  # .MEMD.2756_10 = VDEF <.MEMD.2756_9>
  bD.1872[iD.1875_13] = D.2755_5;
  if (iD.1875_3 <= 149)
    goto <bb 4>;
  else
    goto <bb 5>;
  # SUCC: 4 [99.0%]  (true,exec) 5 [1.0%]  (false,exec)

  # BLOCK 4 freq:9800
  # PRED: 3 [99.0%]  (true,exec)
  goto <bb 3>;
  # SUCC: 3 [100.0%]  (fallthru,dfs_back,exec)

  # BLOCK 5 freq:100
  # PRED: 3 [1.0%]  (false,exec)
  return;
  # SUCC: EXIT [100.0%]

}
```

A.2 Variable distance data dependence

Given below is the code and corresponding result of data dependence pass of gcc on loop with variable distance data dependence.

Code

```
#include <stdio.h>
int a[200],b[200];
```

```
void main()
{
   int i,j;
   for (i=0; i<150; i++)
   {
a[i + 1] = i;
b[i] = a[i + 6] + i;
   }
}
```

Output of data dependence pass of gcc:

```
;; Function main (main)

Creating dr for a[D.2113_4]
analyze_innermost: (analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = &a)
(get_scalar_evolution
  (scalar = &a)
  (scalar_evolution = ))
)
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = (unsigned int) D.2113_4 * 4)
(get_scalar_evolution
  (scalar = (unsigned int) D.2113_4 * 4)
  (scalar_evolution = ))
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = (unsigned int) D.2113_4)
(get_scalar_evolution
  (scalar = (unsigned int) D.2113_4)
  (scalar_evolution = ))
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = D.2113_4)
(get_scalar_evolution
  (scalar = D.2113_4)
  (scalar_evolution = ))
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = D.2112_3)
(get_scalar_evolution
  (scalar = D.2112_3)
  (scalar_evolution = ))
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = i_20)
(get_scalar_evolution
  (scalar = i_20)
  (scalar_evolution = ))
(analyze_initial_condition
  (loop_phi_node =
i_20 = PHI <i_10(4), 0(2)>
)
  (init_cond = 0))
(analyze_evolution_in_loop
  (loop_phi_node = i_20 = PHI <i_10(4), 0(2)>
)
(add_to_evolution
  (loop_nb = 1)
  (chrec_before = 0)
  (to_add = 1)
  (res = {0, +, 1}_1))
  (evolution_function = {0, +, 1}_1))
(set_scalar_evolution
  instantiated_below = 2
  (scalar = i_20)
  (scalar_evolution = {0, +, 1}_1))
)
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = 2)
(get_scalar_evolution
  (scalar = 2)
  (scalar_evolution = 2))
)
(set_scalar_evolution
  instantiated_below = 2
  (scalar = D.2112_3)
  (scalar_evolution = {0, +, 2}_1))
)
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = 1)
(get_scalar_evolution
  (scalar = 1)
  (scalar_evolution = 1))
)
(set_scalar_evolution
  instantiated_below = 2
  (scalar = D.2113_4)
  (scalar_evolution = {1, +, 2}_1))
)
)
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = 4)
(get_scalar_evolution
  (scalar = 4)
  (scalar_evolution = 4))
)
)
success.
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = D.2113_4)
(get_scalar_evolution
  (scalar = D.2113_4)
  (scalar_evolution = {1, +, 2}_1))
(set_scalar_evolution
  instantiated_below = 2
  (scalar = D.2113_4)
  (scalar_evolution = {1, +, 2}_1))
)
(instantiate_scev
  (instantiate_below = 2)
  (evolution_loop = 1)
  (chrec = {1, +, 2}_1)
  (res = {1, +, 2}_1))
base_address: &a
offset from base address: 0
constant offset from base address: 4
step: 8
aligned to: 128
base_object: a[0]
symbol tag: a
Creating dr for a[D.2116_7]
analyze_innermost: (analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = &a)
(get_scalar_evolution
  (scalar = &a)
  (scalar_evolution = ))
)
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = (unsigned int) D.2116_7 * 4)
(get_scalar_evolution
  (scalar = (unsigned int) D.2116_7 * 4)
  (scalar_evolution = ))
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = (unsigned int) D.2116_7)
(get_scalar_evolution
  (scalar = (unsigned int) D.2116_7)
  (scalar_evolution = ))
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = D.2116_7)
(get_scalar_evolution
  (scalar = D.2116_7)
  (scalar_evolution = ))
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = D.2115_6)
(get_scalar_evolution
  (scalar = D.2115_6)
  (scalar_evolution = ))
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = i_20)
(get_scalar_evolution
  (scalar = i_20)
  (scalar_evolution = {0, +, 1}_1))
(set_scalar_evolution
  instantiated_below = 2
  (scalar = i_20)
  (scalar_evolution = {0, +, 1}_1))
)
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = 3)
(get_scalar_evolution
  (scalar = 3)
  (scalar_evolution = 3))
)
(set_scalar_evolution
  instantiated_below = 2
  (scalar = D.2115_6)
  (scalar_evolution = {0, +, 3}_1))
)
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = 6)
(get_scalar_evolution
  (scalar = 6)
  (scalar_evolution = 6))
)
(set_scalar_evolution
  instantiated_below = 2
  (scalar = D.2116_7)
  (scalar_evolution = {6, +, 3}_1))
)
)
```

```
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = 4)
(get_scalar_evolution
  (scalar = 4)
  (scalar_evolution = 4))
)
)
success.
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = D.2116_7)
(get_scalar_evolution
  (scalar = D.2116_7)
  (scalar_evolution = {6, +, 3}_1))
(set_scalar_evolution
  instantiated_below = 2
  (scalar = D.2116_7)
  (scalar_evolution = {6, +, 3}_1))
)
(instantiate_scev
  (instantiate_below = 2)
  (evolution_loop = 1)
  (chrec = {6, +, 3}_1)
  (res = {6, +, 3}_1))
base_address: &a
offset from base address: 0
constant offset from base address: 24
step: 12
aligned to: 128
base_object: a[0]
symbol tag: a
Creating dr for b[i_20]
analyze_innermost: (analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = &b)
(get_scalar_evolution
  (scalar = &b)
  (scalar_evolution = ))
)
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = (unsigned int) i_20 * 4)
(get_scalar_evolution
  (scalar = (unsigned int) i_20 * 4)
  (scalar_evolution = ))
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = (unsigned int) i_20)
```

```
(get_scalar_evolution
  (scalar = (unsigned int) i_20)
  (scalar_evolution = ))
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = i_20)
(get_scalar_evolution
  (scalar = i_20)
  (scalar_evolution = {0, +, 1}_1))
(set_scalar_evolution
  instantiated_below = 2
  (scalar = i_20)
  (scalar_evolution = {0, +, 1}_1))
)
)
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = 4)
(get_scalar_evolution
  (scalar = 4)
  (scalar_evolution = 4))
)
)
success.
(analyze_scalar_evolution
  (loop_nb = 1)
  (scalar = i_20)
(get_scalar_evolution
  (scalar = i_20)
  (scalar_evolution = {0, +, 1}_1))
(set_scalar_evolution
  instantiated_below = 2
  (scalar = i_20)
  (scalar_evolution = {0, +, 1}_1))
)
(instantiate_scev
  (instantiate_below = 2)
  (evolution_loop = 1)
  (chrec = {0, +, 1}_1)
  (res = {0, +, 1}_1))
base_address: &b
offset from base address: 0
constant offset from base address: 0
step: 4
aligned to: 128
base_object: b[0]
symbol tag: b
(compute_affine_dependence
  (stmt_a =
a[D.2113_4] = i_20;
)
  (stmt_b =
D.2117_8 = a[D.2116_7];
)
(subscript_dependence_tester
(analyze_overlapping_iterations
  (chrec_a = {1, +, 2}_1)
  (chrec_b = {6, +, 3}_1)
(analyze_siv_subscript
(analyze_subscript_affine_affine
  (overlaps_a = [4 + 3 * x_1]
)
  (overlaps_b = [1 + 2 * x_1]
)
)
)
  (overlap_iterations_a = [4 + 3 * x_1]
)
  (overlap_iterations_b = [1 + 2 * x_1]
)
)
(Dependence relation cannot be represented by distance vector.)
)


Banerjee Analyzer
(Data Dep:
(Data Ref:
  stmt: a[D.2113_4] = i_20;
  ref: a[D.2113_4];
  base_object: a[0];
  Access function 0: {1, +, 2}_1
)
(Data Ref:
  stmt: D.2117_8 = a[D.2116_7];
  ref: a[D.2116_7];
  base_object: a[0];
  Access function 0: {6, +, 3}_1
)
  access_fn_A: {1, +, 2}_1
  access_fn_B: {6, +, 3}_1

 (subscript
  iterations_that_access_an_element_twice_in_A: [4 + 3 * x_1]
  last_conflict: 20
  iterations_that_access_an_element_twice_in_B: [1 + 2 * x_1]
  last_conflict: 20
  (Subscript distance: scev_not_known;
  )
 )
  inner loop index: 0
  loop nest: (1 )
)
Data ref a:
(Data Ref:
  stmt: a[D.2113_4] = i_20;
  ref: a[D.2113_4];
  base_object: a[0];
  Access function 0: {1, +, 2}_1
)
Data ref b:
(Data Ref:
  stmt: D.2117_8 = a[D.2116_7];
  ref: a[D.2116_7];
  base_object: a[0];
  Access function 0: {6, +, 3}_1
)
affine dependence test not usable: access function not affine or constant.
(dependence classified: scev_not_known)
)
(compute_affine_dependence
  (stmt_a =
a[D.2113_4] = i_20;
)
  (stmt_b =
b[i_20] = D.2118_9;
)
)
(compute_affine_dependence
  (stmt_a =
D.2117_8 = a[D.2116_7];
)
  (stmt_b =
b[i_20] = D.2118_9;
)
)
Dependence tester statistics:
Number of dependence tests: 1
Number of dependence tests classified dependent: 1
Number of dependence tests classified independent: 0
Number of undetermined dependence tests: 1
Number of subscript tests: 1
Number of undetermined subscript tests: 0
Number of same subscript function: 0
Number of ziv tests: 0
Number of ziv tests returning dependent: 0
Number of ziv tests returning independent: 0
Number of ziv tests unimplemented: 0
Number of siv tests: 1
Number of siv tests returning dependent: 1
Number of siv tests returning independent: 0
Number of siv tests unimplemented: 0
Number of miv tests: 0
Number of miv tests returning dependent: 0
Number of miv tests returning independent: 0
Number of miv tests unimplemented: 0
(Data Dep:
    (don't know)
)
(Data Dep:
(Data Ref:
  stmt: a[D.2113_4] = i_20;
  ref: a[D.2113_4];
  base_object: a[0];
  Access function 0: {1, +, 2}_1
)
(Data Ref:
  stmt: b[i_20] = D.2118_9;
  ref: b[i_20];
  base_object: b[0];
  Access function 0: {0, +, 1}_1
)
    (no dependence)
)
(Data Dep:
(Data Ref:
  stmt: D.2117_8 = a[D.2116_7];
  ref: a[D.2116_7];
  base_object: a[0];
  Access function 0: {6, +, 3}_1
)
(Data Ref:
  stmt: b[i_20] = D.2118_9;
  ref: b[i_20];
  base_object: b[0];
  Access function 0: {0, +, 1}_1
)
    (no dependence)
)




(classify_chrec {0, +, 1}_1
  affine_univariate
)
```

```
(classify_chrec {0, +, 2}_1
  affine_univariate
)
(classify_chrec {1, +, 2}_1
  affine_univariate
)
(classify_chrec {0, +, 3}_1
  affine_univariate
)
(classify_chrec {6, +, 3}_1
  affine_univariate
)

(
-----------------------------------------
5 affine univariate chrecs
0 affine multivariate chrecs
0 degree greater than 2 polynomials
0 chrec_dont_know chrecs
-----------------------------------------
5 total chrecs
0 with undetermined coefficients
-----------------------------------------
5 chrecs in the scev database
10 sets in the scev database
26 gets in the scev database
-----------------------------------------
)


Pass statistics:
----------------

main ()
{
  intD.0 pretmp.29D.2132;
  intD.0 iD.2107;
  intD.0 D.2118;
  intD.0 D.2117;
  intD.0 D.2116;
  intD.0 D.2115;
  intD.0 D.2113;
  intD.0 D.2112;

  # BLOCK 2 freq:100
  # PRED: ENTRY [100.0%]  (fallthru,exec)
  # SUCC: 3 [100.0%]  (fallthru,exec)

  # BLOCK 3 freq:9900
  # PRED: 4 [100.0%]  (fallthru,dfs_back,exec) 2 [100.0%]  (fallthru,exec)
  # iD.2107_20 = PHI <iD.2107_10(4), 0(2)>
  # aD.2103_21 = PHI <aD.2103_15(4), aD.2103_13(D)(2)>
  # bD.2104_22 = PHI <bD.2104_16(4), bD.2104_14(D)(2)>
  D.2112_3 = iD.2107_20 * 2;
  D.2113_4 = D.2112_3 + 1;
  # aD.2103_15 = VDEF <aD.2103_21> { aD.2103 }
  aD.2103[D.2113_4] = iD.2107_20;
  D.2115_6 = iD.2107_20 * 3;
  D.2116_7 = D.2115_6 + 6;
  # VUSE <aD.2103_15> { aD.2103 }
  D.2117_8 = aD.2103[D.2116_7];
  D.2118_9 = D.2117_8 + iD.2107_20;
  # bD.2104_16 = VDEF <bD.2104_22> { bD.2104 }
  bD.2104[iD.2107_20] = D.2118_9;
  iD.2107_10 = iD.2107_20 + 1;
  if (iD.2107_10 <= 149)
    goto <bb 4>;
  else
    goto <bb 5>;
  # SUCC: 4 [99.0%]  (true,exec) 5 [1.0%]  (false,exec)

  # BLOCK 4 freq:9800
  # PRED: 3 [99.0%]  (true,exec)
  goto <bb 3>;
  # SUCC: 3 [100.0%]  (fallthru,dfs_back,exec)

  # BLOCK 5 freq:100
  # PRED: 3 [1.0%]  (false,exec)
  return;
  # SUCC: EXIT [100.0%]

}
```