

Efficient Polynomial-Time Nested Loop Fusion with Full Parallelism*

Edwin H.-M. Sha
Dept. of Computer Science
Erik Jonsson School of Eng. & C.S.
Box 830688, MS EC 31
Univ. of Texas at Dallas
Richardson, TX 75083-0688

Timothy W. O'Neil
Dept. of Comp. Science & Eng.
University of Notre Dame
Notre Dame, IN 46556

Nelson L. Passos
Dept. of Computer Science
Midwestern State University
Wichita Falls, TX 76308

Abstract

Data locality and synchronization overhead are two important factors that affect the performance of applications on multiprocessors. Loop fusion is an effective way for reducing synchronization and improving data locality. Traditional fusion techniques, however, either cannot address the case when *fusion-preventing* dependencies exist in nested loops, or cannot achieve good parallelism after fusion. This paper presents a significant addition to the current loop fusion techniques by presenting several efficient polynomial-time algorithms to solve these problems. These algorithms, based on multi-dimensional retiming, allow nested loop fusion even in the presence of outmost loop-carried dependencies or fusion-preventing dependencies. The multiple loops are modeled by a multi-dimensional *loop dependence graph*. The algorithms are applied to such a graph in order to perform the fusion and to obtain full parallelism in the innermost loop.

Key Words: Loop Fusion, Loop Transformation, Nested Loops, Retiming, Data Locality

*This work is partially supported by NSF grants MIP-95-01006 and MIP-97-04276, and by the W. D. Mensch Jr. and A. J. Schmitt Foundations.

1 Introduction

Multi-dimensional applications such as image processing, fluid mechanics and weather forecasting may deal with billions of data values, requiring high performance computers. Computation intensive sections of those applications usually consist of the repeated execution of operations coded as nested loops. Consecutive loops in a program are responsible for computational overhead due to repetitive access of data elements residing in non-local memories, as well as for synchronization between processors in a parallel environment. Loop transformations offer effective ways to improve parallelism and achieve efficient use of the memory, while loop partitioning is valuable in improving data locality and reducing communication. Loop fusion is a loop-reordering transformation that merges multiple loops into a single one. However, the existing fusion techniques do not consider the implementation of loop fusion with multi-dimensional dependencies. In this paper, a new technique which performs the loop fusion even in the presence of fusion-preventing dependencies and allows for the achievement of a fully parallel execution of the innermost fused loop is developed.

Loop transformations such as loop interchange, loop permutation, loop skewing, loop reversal and loop tiling are some of the techniques used to optimize the execution of loops with respect to processing speed (parallelism) and memory access [6, 19, 27, 31–33]. Data locality and communication between processing elements can also be optimized by using partitioning techniques [1, 2, 12, 28]. These techniques are usually applied to single loops and do not target the simultaneous optimization of multiple loops. Loop fusion seeks to combine multiple loops into a single one, allowing the use of more traditional optimization methods [4, 5, 11]. By increasing the granule size of the loops, it is possible to improve data locality and to reduce synchronization [3, 14]. This paper presents a new technique able to fuse the loops while improving the parallel execution of the resulting code.

Warren [30] discusses the conditions that prevent loop fusion. Researchers in this area are focusing on solutions to problems where the fusion is not always legal either because of the existence of fusion-preventing dependencies between loops (i.e. dependencies that would become reverse to the computational flow after the fusion) or because the fusion may introduce loop-carried dependencies and thus reduce loop parallelism [5, 33, 35]. Kennedy and McKinley [13, 14] perform loop fusion to combine a collection of loops and use loop distribution to improve parallelism. However they do not address the case when fusion-preventing dependencies exist.

Carr [7] presents a loop fusion algorithm based on a cost model, but he also does not con-

sider fusion-preventing dependencies and so no full parallelism is guaranteed. Al-Mouhamed's [3] method of loop fusion is based on vertical and horizontal fusion, with fusion not performed if fusion-preventing dependencies exist or if the fused loop prevents parallelization. Manjikian and Abdelrahman [17, 18] suggest a "shift-and-peel" loop transformation to fuse loops and allow parallel execution. The shifting part of the transformation may fuse loops in the presence of fusion-preventing dependencies. However, when the number of peeled iterations exceeds the number of iterations per processor, this method is not efficient.

Affine-scheduling methods focus on improving parallelism in the entire program body [9, 10, 16, 29]. Loop fusion then becomes a side effect of applying such a method to code with consecutive loop segments. Examples of such methods are the affine-by-statement and affine scheduling techniques described by Dart and Robert [9] and Feautrier [10], which offer a general solution to the parallel optimization problem. These solutions usually depend on linear programming techniques to achieve the optimized results and are directly applicable to hardware design. The solution presented in this paper utilizes the multi-dimensional retiming technique, which is applicable to both hardware and software implementations [22, 23].

The multi-dimensional retiming methodology has been shown to provide the necessary fundamentals for improving parallelism in systems with a limited number of processors [25], and also to allow simultaneous optimization of local memory utilization [21]. The implementation of the multi-dimensional retiming technique on VLSI systems also has been discussed in [24] and shown to be less complex than other techniques. In this paper, we focus on the loop fusion problem, concentrating on a comparison of the proposed solution with similar techniques recently published [3, 7, 13, 17, 18]. To the authors' knowledge, most work on loop fusion has not addressed problems characterized by nested loop (multi-level) fusion-preventing carried dependencies. In this paper, these problems are solved using the idea of multi-dimensional retiming [15, 20, 25, 26]. By using multi-dimensional retiming, the proposed technique can do legal loop fusion even in the case of fusion-preventing dependencies, achieving full parallelism of the innermost loop.

The target problem is represented by the code shown in Figure 1, where the inner loop instructions were omitted and the innermost loops are DOALL loops that work in the same range of control indices. The program contains only fully parallel innermost loops and data dependencies with constant distances [34]. We model the problem by a *multi-dimensional loop dependence graph (MLDG)*, where each innermost loop is represented as a node. Instead of simply fusing the candidate loop

nests directly, the loops are rescheduled by applying multi-dimensional retiming to the MLDG. A new MLDG is constructed through the retiming function such that no fusion-preventing dependencies exist after retiming and the loop fusion is legal.

```

DO k1 i = 0, n
    DOALL k2 j = 0, m
    ....
k2 CONTINUE
    DOALL k3 j = 0, m
    ....
k3 CONTINUE
    DOALL k4 j = 0, m
    ....
k4 CONTINUE
    DOALL k5 j = 0, m
    ....
k5 CONTINUE
    .
    .
    .
k1 CONTINUE

```

Figure 1: Our program model

This paper focuses on two-dimensional cases, presenting three polynomial-time algorithms aimed at handling different situations found in loop fusion. The first algorithm deals with the case when no cycle exists in the MLDG. A second situation involves the existence of cycles. In this case, it is necessary to satisfy certain restrictions in order to obtain full parallelism after applying the proposed fusion technique. The third case consists of a general solution where, after retiming and loop fusion, a hyperplane is computed such that all iterations comprising the innermost loop can be executed in parallel [20].

Figure 2(a) shows an example of a two-dimensional loop dependence graph (*2LDG*) consisting of 4 nodes, equivalent to the 4 innermost loops of the code of Figure 2(b). The nodes in the graph represent each of the DOALL loops according to the labels A, B, C and D. Edges between two nodes show that there exists a data dependence between them. For example, data produced in loop A will be consumed within loop B. The edge weight shows a multi-dimensional offset between the data production and consumption. Applying our new technique we obtain the retimed graph presented in Figure 3(a). Figure 3(b) shows the pseudo code after retiming and loop fusion. It is easy to see that the resulting innermost loop is fully parallel. An initial sequence (the *prologue*) is created in order to provide the correct initial data. Such additional code usually requires a small computation time when compared to that of the total execution of the innermost loop and can be considered negligible. The resulting fused loop has a lower number of synchronization requests than the initial group of four loops, reducing the synchronization overhead.

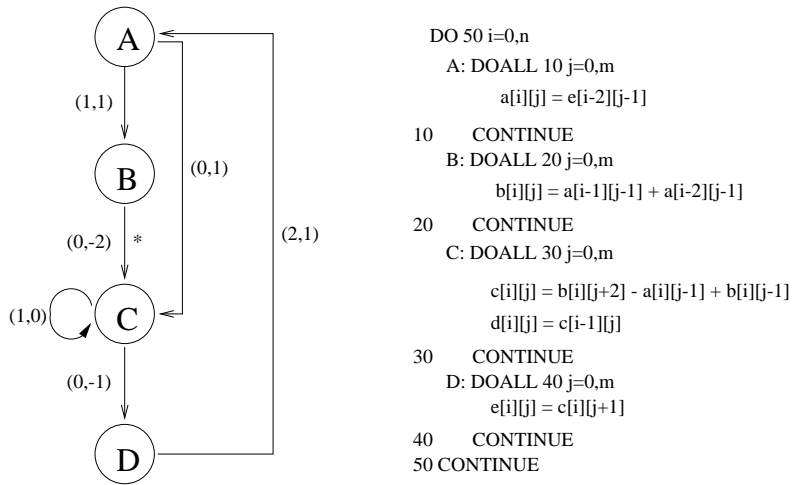


Figure 2: (a) An example of a 2LDG; (b) The equivalent code

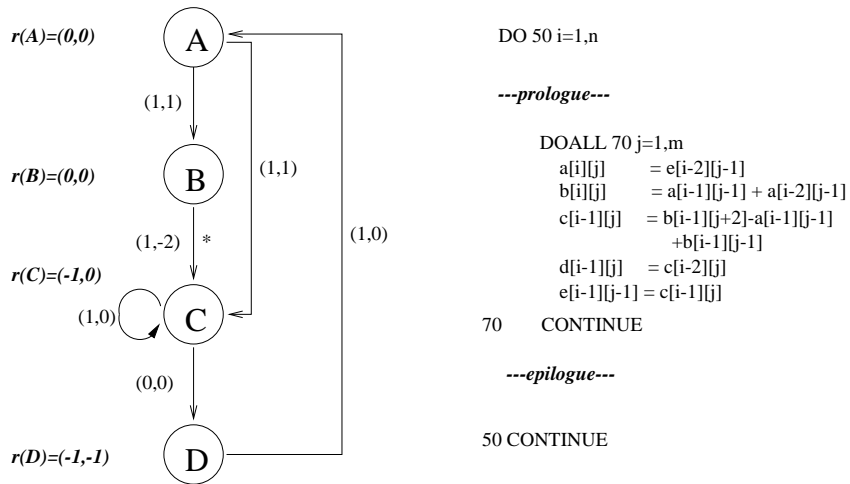


Figure 3: (a) Our 2LDG after retiming; (b) The pseudocode after retiming and fusion

The details of the transformation are presented in this paper, beginning with an introduction to the basic concepts in the next section, in which we formally describe the graph model for fusion and retiming. In Section 3, we show that it is always possible to find a retiming function which makes the loop fusion legal. We discuss cases where the full parallelism of the innermost loop is possible under the same original execution sequence in Section 4. Detailed examples are discussed in Section 5, and then a conclusion is drawn to complete the paper.

2 Basic Concepts

Loop fusion combines adjacent loop bodies into a single body. Because of array reuse, it reduces the references to main memory. It also reduces the synchronization between parallel loops. However, fusion is not always legal. For example, let S_1 and S_2 be two statements in two different candidate loops. If the calculation of S_2 depends on the result from S_1 , but after loop fusion the execution of S_2 comes earlier than that of S_1 , the loop fusion is illegal [17, 33]. In other words, if the direction of the data dependence becomes the reverse of the control flow, loop fusion is illegal. Figure 4(a) is the 2LDG of the code in Figure 2, and Figure 4(b) shows the code after an illegal loop fusion. In this example, $c[i][j]$ depends on $b[i][j+2]$, but $b[i][j+2]$ has not been calculated yet. An adequate analysis of these data dependencies is required before fusing the loops. In order to develop an algorithm for solving the loop fusion problem, we model the nested loop as a graph. The following subsections present the basic concepts necessary for developing the new algorithm.

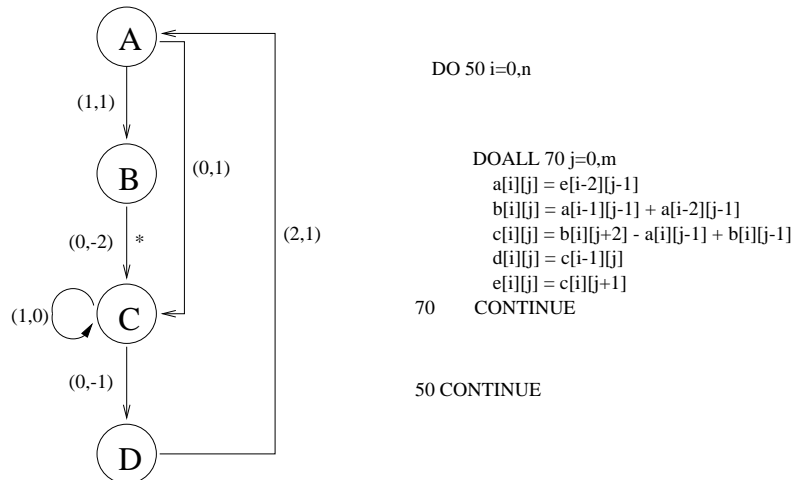


Figure 4: (a) Our original 2LDG; (b) Our original code after illegal loop fusion

2.1 Data Dependencies

Data dependence is the most important factor to be considered in the loop fusion. Dependencies between two statements in a program commonly mentioned in the literature include *true dependence*, *antidependence* and *output dependence* [4, 14]. In this paper, new terms such as *outmost loop-carried dependence* and *self-dependence* are considered.

An *outmost loop-carried dependence* occurs when one statement stores a data value on one iteration, and that value is fetched on a later iteration of the outmost loop. *Self-dependence* is a special case of outmost loop-carried dependence where the data is produced and consumed by the same innermost loop. This is a small abuse of the traditional definition, which applies to the case where a statement depends upon itself. For example, in Figure 2, we will consider the dependence between the computation of c and its consumption in loop C to be a case of self-dependence, even though production and consumption do not occur in the same statement. On the other hand, the dependence between the production of variable e in loop D and its consumption in loop A is a case of loop-carried dependence. Later it will be shown that the fusion in the presence of outmost loop-carried dependencies is always legal. When one statement stores a data value and another statement uses that value in the same iteration of the outmost loop, the fusion may not always be legal. This is the case of a *fusion-preventing dependence*. Fusion-preventing dependencies are dependencies that become reverse to the control flow after fusion, making the fusion illegal.

Disregarding the dependencies involved, the iterative execution of the loop can be represented as an iteration space. An *iteration space* is a Cartesian space, where each iteration is represented by a point (i, j) such that i represents the index of the outmost loop and j represents the index of the innermost loop. In our model, there is one iteration space associated with every innermost loop. For example, in Figure 2(b) there are four iteration spaces. Loop fusion combines the iteration spaces into one single space.

Because of array reuse, data dependencies may exist between the different iteration spaces. A loop dependence vector is used to represent such dependencies.

Definition 2.1 *Given two iteration spaces representing loops A and B sequentially executed within an outermost loop, with iteration $(i_1, j_1) \in B$ and iteration $(i_2, j_2) \in A$, then we say that there is a loop dependence vector $d_L = (k_1, k_2) = (i_1 - i_2, j_1 - j_2)$ that represents the data dependence between those loops if a data value is computed in (i_2, j_2) and consumed at (i_1, j_1) .*

In Figure 2(b), there is a loop dependence vector $(0, -2)$ between loops B and C, and there are loop dependence vectors $(1, 1)$ and $(2, 1)$ between loops A and B. These loop dependence vectors provide important information for producing a valid loop fusion.

Finally, given a set S of loop-dependence vectors, we will say that v is the *minimal loop dependence vector* in S if $v \in S$ and $v \leq u$ for all other vectors $u \in S$ according to a lexicographic order [31]. In the two-dimensional case, a vector $v = (a, b)$ is smaller than a vector $u = (x, y)$ according to lexicographic order if either $a < x$ or $a = x$ and $b < y$. This idea will prove to be important in the next subsection when we formally define MLDGs.

2.2 Problem Model

We model the multiple loops problem as a graph in which each innermost loop is represented by a node and the data dependencies between loops are represented as directed edges between the nodes. For example, in Figure 2, two of the existing loop dependence vectors between loops A and B are $(1,1)$ and $(2,1)$, and so there is an edge in our graph from node A to node B with weight $(1,1)$, the smaller of the two vectors. Such a graph is called a *multi-dimensional loop dependence graph* as defined below:

Definition 2.2 *Given a nested loop with multiple DOALL innermost loops, a multi-dimensional loop dependence graph (MLDG) $G = (V, E, \delta_L, D_L)$ having dimension n is a node-weighted and edge-weighted directed graph, where:*

- V is the set of nodes representing the innermost loop nests,
- $E \subseteq V \times V$ is the set of edges representing dependencies between the loops; if nodes a and b represent loops A and B , respectively, and loop B uses one or more results from loop A , then there is one edge from a to b in G ,
- D_L is a function from $V \times V$ to $\wp(Z^n)$ which returns the set of all loop dependence vectors between two connected nodes,
- δ_L is a function from E to Z^n which returns the minimal loop dependency vector of an edge; in other words, given an edge $e : a \rightarrow b$, $\delta_L(e) = \min\{v : v \in D_L(a, b)\}$.

Our algorithm works with the minimal loop dependence vector function designated $\delta_L(e)$. Therefore, an MLDG can be reduced to a graph where there exists at most one edge from one node to any other, labelled with the minimal loop dependence vector. For example, Figure 2 is an example of an MLDG. In this example, $V = \{A, B, C, D\}$ and $E = \{e_1 : A \rightarrow B, e_2 : B \rightarrow C, e_3 : C \rightarrow D, e_4 : A \rightarrow C, e_5 : D \rightarrow A, e_6 : C \rightarrow C\}$ where $D_L(A, B) = \{(1, 1), (2, 1)\}$, $D_L(B, C) = \{(0, -2), (0, 1)\}$, $D_L(C, D) = \{(0, -1)\}$, $D_L(A, C) = \{(0, 1)\}$, $D_L(D, A) = \{(2, 1)\}$, and $D_L(C, C) = \{(1, 0)\}$; while $\delta_L(e_1) = (1, 1)$, $\delta_L(e_2) = (0, -2)$, $\delta_L(e_3) = (0, -1)$, $\delta_L(e_4) = (0, 1)$, $\delta_L(e_5) = (2, 1)$, and $\delta_L(e_6) = (1, 0)$.

The particular case when there are two or more different loop dependence vectors between two nodes which have the same first coordinates but different second coordinates is also very important to this new technique. Such a case will introduce some constraints in achieving a fully parallel solution after fusion. In order to handle such constraints, we say that such edges are called *parallelism hard edges*, or *hard-edges* for short. We use the symbol $\xrightarrow{*}$ to represent a hard-edge in an MLDG. In Figure 2(a), the edge from B to C is a hard-edge because the loop dependence vectors $(0, -2)$ and $(0, 1)$ between B and C agree on first coordinates but differ on the second. On the other hand, the edge from A to B is not a hard-edge because the loop dependence vectors for that edge, $(1, 1)$ and $(2, 1)$, have different first coordinates.

An MLDG is *legal* if there is no outmost loop-carried dependence vector reverse to the computational flow, i.e., the nested loop is executable. For any cycle c in an MLDG, $\delta_L(c)$ is defined as $\sum_{e_i \in c} \delta_L(e_i)$. The cycles in a legal two-dimensional MLDG are characterized in the lemma below.

Lemma 2.1 *Let $G=(V, E, \delta_L, D_L)$ be a legal 2LDG, i.e. a legal two-dimensional MLDG. For each cycle $c = v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_n$ ($n \geq 2$) in G , $\delta_L(c) \geq (1, -\infty)$.*

Proof: In any legal MLDG, each cycle must contain at least one edge representing an outmost loop-carried dependence. By definition, we know that each outmost loop-carried dependence vector is greater than or equal to $(1, -\infty)$. Now, for each edge e in a cycle, $\delta_L(e) \geq (0, -\infty)$. Furthermore, in each cycle, there is at least one edge e' with outmost loop-carried dependence vector which satisfies $\delta_L(e') \geq (1, -\infty)$. Thus, for each cycle c in G , $\delta_L(c) \geq (1, -\infty)$. \square

For example, in Figure 2(a), cycle $c_1 = A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$ has $\delta_L(c_1) = (3, -1)$, and cycle $c_2 = A \rightarrow C \rightarrow D \rightarrow A$ has $\delta_L(c_2) = (2, 1)$, showing that the 2LDG is legal.

2.3 Multi-Dimensional Retiming

Passos and Sha [20] proposed multi-dimensional retiming techniques that obtain full parallelism in nested loops. The multi-dimensional retiming concept is helpful in solving the loop fusion problem, so we will review some of those concepts below.

A *schedule vector* s is the normal vector for a set of parallel equitemporal hyperplanes that define a sequence of execution of the iterations. We say that a schedule vector s is a *strict schedule vector* for an MLDG $G = (V, E, \delta_L, D_L)$ if, for each edge $e : a \rightarrow b$ and non-zero loop dependence vector $d_L(e) \in D_L(a, b)$, $d_L(e) \cdot s > 0$. For example, $s = (1, 0)$ is a strict schedule vector for the example of Figure 3(a).

A *two-dimensional retiming* r of a 2LDG $G = (V, E, \delta_L, D_L)$ is a function from V to Z^2 that transforms the iteration spaces of the loops. A new 2LDG $G_r = (V, E, \delta_{Lr}, D_{Lr})$ is created such that each outmost iteration still has one execution of each operation belonging to each of the nodes in V . The retiming vector $r(u)$ of a node $u \in V$ represents the offset between the original iteration space representing the loop u and the one after retiming. The loop dependence vectors change accordingly to preserve dependencies, i.e. $r(u)$ counts how many of δ_L 's components are pushed into the edges $u \rightarrow v$ and subtracted from the edges $w \rightarrow u$, where $u, v, w \in V$. Therefore, we have $\delta_{Lr}(e) = \delta_L(e) + r(u) - r(v)$ and $D_{Lr}(u, v) = \{d_L(e) + r(u) - r(v) : d_L(e) \in D_L(u, v)\}$ for each edge $e : u \rightarrow v$. Also, $\delta_{Lr}(c) = \delta_L(c)$ for each cycle $c \in G$. A multi-dimensional retiming may require a change in the schedule vector.

For example, in Figure 3(a), the retiming function for the four nodes is $r(A) = (0, 0)$, $r(B) = (0, 0)$, $r(C) = (-1, 0)$ and $r(D) = (-1, -1)$. After retiming, the weight of edge $e_5 : D \rightarrow A$ becomes $\delta_{Lr}(e_5) = \delta_L(e_5) + r(D) - r(A) = (2, 1) + (-1, -1) - (0, 0) = (1, 0)$, and the set of loop dependence vectors of edge e_5 becomes $D_{Lr}(D, A) = \{(1, 0)\}$. The weights of the cycles remain unchanged: $\delta_{Lr}(c_1) = (3, -1) = \delta_L(c_1)$ for $c_1 = A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$, and $\delta_{Lr}(c_2) = (2, 1) = \delta_L(c_2)$ for $c_2 = A \rightarrow C \rightarrow D \rightarrow A$.

2.4 Two Dimensional Linear Inequality Systems

Our overall goal is to find a multi-dimensional retiming function which results in fused loops when applied to our original code. To accomplish this, we wish to use a special ILP model described in [8]

which can be efficiently solved in polynomial-time to find said function. Therefore, we now present a brief review of this problem.

Problem ILP Given a set of m linear inequalities of the form

$$x_j - x_i \leq a_{ij}$$

on the unknown integers x_1, x_2, \dots, x_n , where a_{ij} are given integer constants, determine feasible values for the unknowns x_i or determine that no such values exist.

To solve this problem, we construct a *constraint graph* $G = (V, E, w)$ as follows:

- $V = \{v_0, v_1, \dots, v_n\}$, where vertex v_i corresponds to the unknown variable x_i for $i = 1, 2, \dots, n$;
- for each constraint $x_j - x_i \leq a_{ij}$, the edge $e : v_i \rightarrow v_j$ is added to E with $w(v_i, v_j) = a_{ij}$;
- for $i = 1, 2, \dots, n$, the edge (v_0, v_i) is added to E with $w(v_0, v_i) = 0$.

The following theorem is from [8].

Theorem 2.2 *An inequality system has a feasible solution if and only if the corresponding constraint graph has no cycle with negative weight.*

Using the Bellman-Ford Algorithm, the shortest path from v_0 to v_i in the constraint graph is a feasible solution x_i of this inequality system. The time complexity of this algorithm is $O(|V||E|)$. Now, we extend this method to two dimensions by stating a new problem.

Problem 2-ILP Given a set of m linear inequalities of the form

$$r_j - r_i \leq w_{ij}$$

on the unknown two-dimensional vectors r_1, r_2, \dots, r_n , where w_{ij} are given two-dimensional vectors, determine feasible values for the unknowns r_i or determine that no such values exist.

The solution is similar to the previous one, in that we construct the corresponding constraint graph $G = (V, E, w)$ as above. The next theorem states the necessary and sufficient conditions for solving this problem. We will use these concepts to develop an algorithm in the coming section which solves the 2-ILP problem.

Theorem 2.3 *An instance of the 2-ILP problem has feasible solution if and only if each cycle of its corresponding constraint graph has weight greater than or equal to $(0, 0)$.*

3 Algorithm for Legal Loop Fusion

In this section, we will demonstrate that it is always possible to find a retiming function that makes loop fusion legal. We then outline an algorithm for computing such a retiming. Later we will build on this method to construct retiming algorithms for more complicated cases. Finally, we will discover a new problem, one which may prevent the parallel execution of the innermost loop nest after retiming.

3.1 Data Dependence Constraints

In order to predict if a loop fusion is legal according to the loop dependence vectors, let us examine a general loop dependence vector $d_L = (d_L[1], d_L[2])$. Three cases may occur:

1. $d_L[1] > 0$. This means that a dependence is carried by the outmost loop. In this case, after loop fusion, the dependence direction is preserved and loop fusion is safe.
2. $d_L[1] = 0, d_L[2] \geq 0$. This implies that a data dependence exists in the same iteration of the outmost loop, and after fusion, the dependence becomes forward loop-carried by the innermost loop or loop-independent. Therefore, the dependence direction is preserved, and the loop fusion is also safe.
3. $d_L[1] = 0, d_L[2] < 0$. This is also a case of dependence in the same iteration of the outmost loop. However, after fusion, the dependence becomes antidependent. The dependence direction is reversed and the straightforward loop fusion is then illegal.

So, if all loop dependence vectors d_L satisfy $d_L \geq (0, 0)$, loop fusion is legal. Based on these three possibilities, the following theorem states the relationship between $\delta_L(e)$ and the loop fusion transformation:

Theorem 3.1 *Given a 2LDG $G = (V, E, \delta_L, D_L)$, if $\delta_L(e) \geq (0, 0)$ for all $e \in E$, then the loop fusion is legal.*

Proof: We know that $\delta_L(e) = \min\{d_L(e) : d_L(e) \in D_L(e)\}$. If $\delta_L(e) \geq (0, 0)$ for an edge e , then all the loop dependence vectors of e are bounded from below by $(0, 0)$. Generalizing to all edges in the graph, we see that all loop dependence vectors are bounded from below by $(0, 0)$, and so the loop fusion is legal. \square

3.2 Loop Fusion By Retiming

Considering the constraints imposed by the data dependence vectors in the loop fusion process, one may conclude that loop fusion is feasible if the loops can be transformed in such a way as to satisfy the dependence constraints of Theorem 3.1. In the following, we will prove that it is always possible to find a retiming function for a legal 2LDG G such that after retiming the weights of its edges, all edges e of the retimed graph $G_r = (V, E, \delta_{Lr}, D_{Lr})$ satisfy $\delta_{Lr}(e) \geq (0, 0)$. We solve this problem by using a 2-dimensional linear inequality system.

Theorem 3.2 *Given a legal 2LDG $G = (V, E, \delta_L, D_L)$, there exists a legal retiming r , such that, after retiming, loop fusion is always legal.*

Proof: Let r be a legal retiming. By definition, $\delta_{Lr}(e) = \delta_L(e) + r(v_i) - r(v_j)$ for each edge $e \in E : v_i \rightarrow v_j$. $\delta_{Lr}(e) \geq (0, 0)$ means that $r(v_j) - r(v_i) \leq \delta_L(e)$. Thus we construct the system

$$r(v_j) - r(v_i) \leq \delta_L(e)$$

with $|E|$ inequalities. If we prove that this inequality system has a feasible solution, it would imply the existence of a legal retiming r such that the weight of each edge after retiming satisfies $\delta_{Lr}(e) \geq (0, 0)$. We know that we can define a constraint graph $G' = (V', E', w')$ for the inequality system such that $V' = V \cup \{v_0 : v_0 \notin V\}$, $E' = E \cup \{v_0 \rightarrow v_i : v_i \in V\}$, $w'(v_i, v_j) = \delta_L(e)$ for all $e : v_i \rightarrow v_j \in E$ and $w'(v_0, v_i) = (0, 0)$ for all $v_i \in V$. From this construction we know that G' has the same cycles as G , and $w'(c') = \delta_L(c)$ for each corresponding pair of cycles $c' \in G'$ and $c \in G$. By Lemma 2.1 all cycles c in G satisfy $\delta_L(c) \geq (1, -\infty)$, and so all cycles c' in G' satisfy $w'(c') \geq (1, -\infty) > (0, 0)$. Therefore the inequality system has a feasible solution by Theorem 2.3, and so we can find the legal retiming r such that loop fusion is legal. \square

3.3 The Algorithm

In our technique we use a two-dimensional version of the Bellman-Ford algorithm to compute the retiming function, which is the shortest path from an initial node v_0 to each other node, as shown in Algorithm 1 below. The ability to predict the retiming function for a legal fusion of any 2LDG then allows us to define Algorithm 2 below.

Figure 5 shows the constraint graph for the example of Figure 2(a). As we can see, node v_0 has been added to the MLDG as well as the edges from v_0 to every other node of the MLDG. The

Algorithm 1 The two-dimensional Bellman-Ford algorithm (TwoDimBellmanFord)

Input: A constraint graph $G' = (V', E', w')$.

Output: A retiming function r .

```
 $r(v_0) \leftarrow (0, 0)$ 
for all nodes  $v \in V' - \{v_0\}$  do
   $r(v) \leftarrow (\infty, \infty)$ 
end for
for  $k \leftarrow 1$  to  $|V'| - 1$  do
  for all edges  $(v_i, v_j) \in E'$  do
    if  $r(v_j) > r(v_i) + w'(v_i, v_j)$  then
       $r(v_j) \leftarrow r(v_i) + w'(v_i, v_j)$ 
    end if
  end for
end for
return  $r$ 
```

Algorithm 2 Legal Loop Fusion Retiming Algorithm (LLOFRA)

Input: A 2LDG $G = (V, E, \delta_L, D_L)$.

Output: A retiming function r of the 2LDG.

```
 $V' \leftarrow V \cup \{v_0\}$  /* Construct the constraint graph  $G' = (V', E', w')$  from  $G$ . */
 $E' \leftarrow E \cup \{v_0 \rightarrow v_i : v_i \in V\}$ 
for all edges  $e \in E$  do
   $w'(e) \leftarrow \delta_L(e)$ 
end for
for all edges  $e' \in E' - E$  do
   $w'(e') \leftarrow (0, 0)$ 
end for
 $r \leftarrow \text{TwoDimBellmanFord}(G')$  /* Use Algorithm 1 to compute the retiming function. */
return  $r$ 
```

retiming function computed by the algorithm above is $r(A) = (0, 0)$, $r(B) = (0, 0)$, $r(C) = (0, -2)$, and $r(D) = (0, -3)$. Using such a function, we obtain the new 2LDG shown in Figure 6(a). Figure 6(b) shows the code for the innermost loop after fusion and retiming.

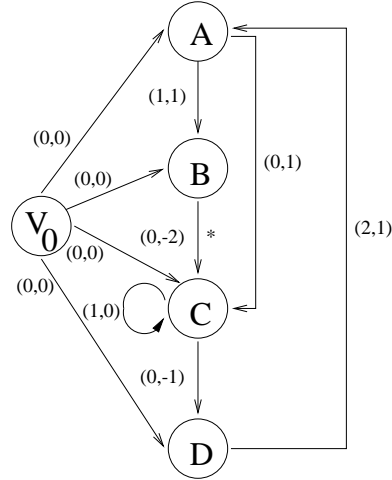


Figure 5: The constraint graph for the 2LDG of Figure 2(a)

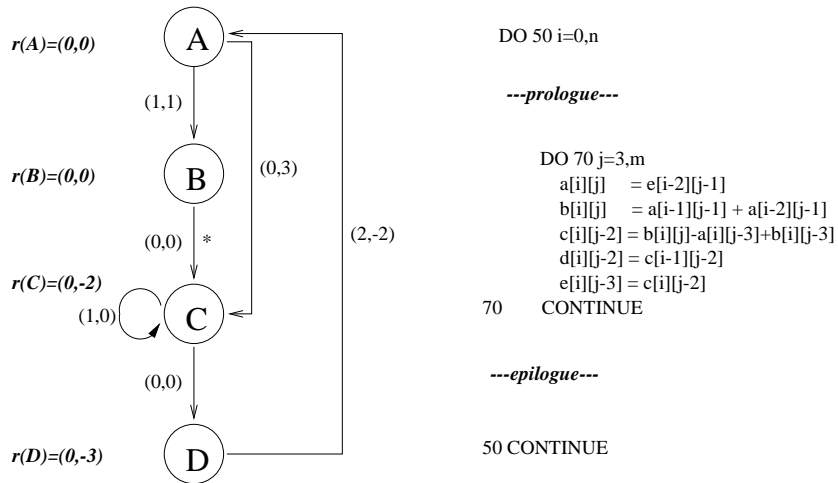


Figure 6: (a) Our 2LDG after retiming; (b) Legally fused pseudocode after retiming

However, after loop fusion the innermost loop may not be executed with full parallelism because of new innermost loop-carried dependences. Figure 7 shows the iteration space for the example in Figure 6. It is obvious that it contains data dependences between the iterations of the innermost loop (a row in the iteration space). Hence, the execution of the innermost loop becomes serial and not fully parallel as desired. However, before loop fusion, the iterations of each candidate loop could

be executed in parallel, and synchronization was only required to ensure that all iterations of one candidate loop would be executed before any iteration of another candidate loop. In the following section, we will discuss how to solve this new problem.

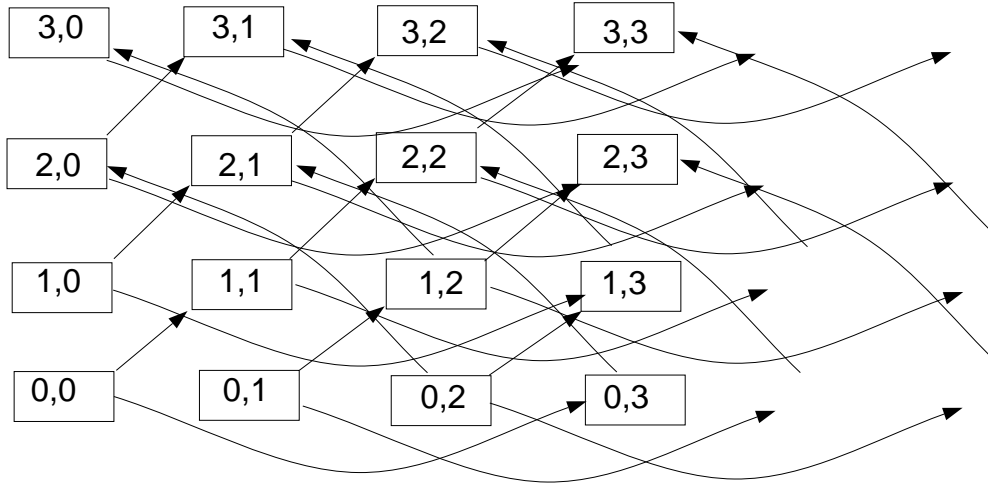


Figure 7: The iteration space after retiming and loop fusion

4 Algorithms for Parallel Loop Fusion

In this section, the algorithms required to solve different cases of loop fusion are presented. Initially, problems that allow the parallel execution of the fused loop according to the original sequence of execution are discussed. Next, fusion preventing conditions introduced by dependence vectors are analyzed and eliminated by using a method similar to a wavefront execution approach resulting from the application of multi-dimensional retiming.

4.1 Properties of DOALL Innermost Loops

We know that fusion is safe if all the dependence vectors are greater than or equal to (0,0). However, there is no guarantee that the parallelism after fusion will be preserved. In this subsection, we discuss those cases in which we can find a retiming function such that the innermost loop can be executed in parallel after retiming and loop fusion. An innermost loop is said to be *DOALL* when all iterations of row i (i.e. iterations $(i, 0), (i, 1), (i, 2), \dots$) can be done in parallel. In other words, an innermost loop

is DOALL if, for any two iterations (i, j_1) and (i, j_2) , there is no data dependence between them, i.e. no dependence vector $(0, k)$, $k \neq 0$, exists.

The following concepts discuss the problem of the parallelism after fusion. We begin by identifying the properties of a DOALL innermost loop with respect to the scheduling vector.

Property 4.1 *Let $G = (V, E, \delta_L, D_L)$ be a 2LDG. If, after loop fusion, the corresponding fused innermost loop nest is a DOALL loop, then it can be executed according to a strict schedule vector $s = (1, 0)$.*

Proof: Let a and b be adjacent nodes. From the definition of a strict schedule vector, $s \cdot d_L(e) > 0$ for all non-zero $d_L(e) \in D_L(a, b)$. Since $s = (1, 0)$, $d_L(e)[1] \neq 0$. Therefore there is no dependence $d_L(e) = (0, k)$, which satisfies the definition of a DOALL loop. \square

Property 4.2 *Let $G = (V, E, \delta_L, D_L)$ be a 2LDG. Let $G_r = (V, E, \delta_{Lr}, D_{Lr})$ be this 2LDG after retiming and loop fusion. If, after loop fusion, the corresponding fused innermost loop nest is a DOALL loop, then all of its dependence vectors satisfy the condition $d_{Lr}(e) \geq (1, -\infty)$ or $d_{Lr}(e) = (0, 0)$ for $e : a \rightarrow b \in E$, $d_{Lr}(e) \in D_{Lr}(a, b)$.*

Proof: Immediate from Property 4.1. \square

The theorems in the next subsections will predict if we can find a legal retiming function to achieve loop fusion and full parallelism. We solve this problem by using the model of integer linear programming and the Bellman-Ford algorithm described in the previous section.

4.2 Fully Parallel Loop Fusion of Acyclic 2LDGs

Theorem 4.1 *Let $G = (V, E, \delta_L, D_L)$ be a 2LDG. If G is acyclic, then there exists a legal retiming r such that, after retiming and loop fusion, the innermost loop nest is DOALL.*

Proof: From the definition of retiming, $\delta_{Lr}(e) = \delta_L(e) + r(v_i) - r(v_j)$ for each edge $e : v_i \rightarrow v_j$. To satisfy the DOALL requirements we need $\delta_{Lr}(e) \geq (1, -\infty)$, which means that $r(v_j) - r(v_i) \leq \delta_L(e) - (1, -\infty)$. We construct the inequality system

$$r(v_j) - r(v_i) \leq \delta_L(e) - (1, -\infty).$$

If this inequality system has a feasible solution then there exists a legal retiming r such that, after retiming, the weight of each edge satisfies $\delta_{Lr}(e) \geq (1, -\infty)$. As in Theorem 3.2, we construct the

constraint graph G' , which has no cycle because G has no cycle. Therefore, by Theorem 2.3 the inequality system has a solution, and by Property 4.2 the innermost loop is DOALL after retiming and loop fusion. \square

Now we modify the LLOFRA algorithm (Algorithm 2, presented in Section 3) in order to obtain a retiming function for legal fusion and full parallelism.

Algorithm 3 Legal Loop Fusion and Full Parallelism for Acyclic 2LDGs

Input: A 2LDG $G = (V, E, \delta_L, D_L)$.

Output: A retiming function r of the 2LDG.

```

 $V' \leftarrow V \cup \{v_0\}$           /* Construct the constraint graph  $G' = (V', E', w')$  from  $G$ . */
 $E' \leftarrow E \cup \{v_0 \rightarrow v_i : v_i \in V\}$ 
for all edges  $e \in E$  do
     $w'(e') \leftarrow \delta_L(e) - (1, -\infty)$ 
end for
for all edges  $e' \in E' - E$  do
     $w'(e') \leftarrow (0, 0)$ 
end for
 $r \leftarrow \text{TwoDimBellmanFord}(G')$     /* Use Algorithm 1 to compute the retiming function. */
for all nodes  $v \in V$  do
     $r(v)[2] \leftarrow 0$                 /* Set the second component of  $r$ . */
end for
return  $r$ 

```

As an example, consider a 2LDG $G = (V, E, \delta_L, D_L)$ with

- $V = A, B, C, D, E, F, G$;
- $E = \{A \rightarrow B, B \rightarrow C, C \rightarrow D, D \rightarrow E, B \rightarrow F, F \rightarrow G, B \rightarrow E, A \rightarrow D\}$;
- $D_L(A, B) = \{(0, 1)\}$, $D_L(B, C) = \{(0, -2), (0, 3)\}$, $D_L(C, D) = \{(1, 3)\}$, $D_L(D, E) = \{(2, -2)\}$, $D_L(B, F) = \{(0, -2)\}$, $D_L(F, G) = \{(1, 2)\}$, $D_L(B, E) = \{(1, 2)\}$, and $D_L(A, D) = \{(0, -3), (0, -1)\}$.

This example is shown in Figure 8. This 2LDG needs 7 synchronizations for each outmost loop iteration. Thus, assuming n as the number of iterations of the outmost loop, we need $7 * n$ synchronizations. Because fusion-preventing dependences such as $(0, -2)$ and $(0, -3)$ exist, we cannot fuse loops directly. Applying Algorithm 3, first the constraint graph is constructed (Figure 9) and then used to compute the retiming function and retime the original graph (Figure 10). The retimed code

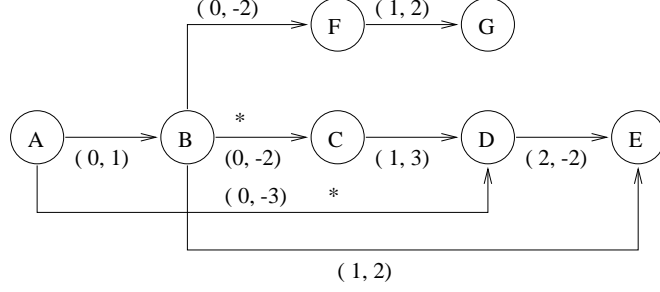


Figure 8: A sample acyclic 2LDG before retiming

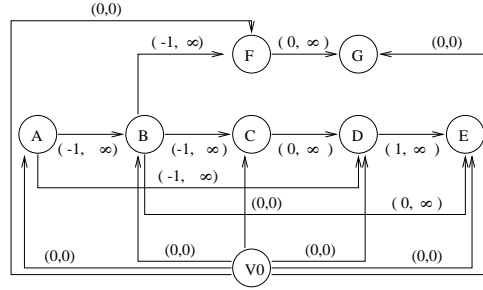


Figure 9: The constraint graph of Figure 8

has no fusion-preventing dependences and thus the loop fusion is legal. The fused loop requires only one synchronization for each outmost loop iteration, for a total of $(n - 2)$ synchronizations. We can also see that the resulting innermost loop can be executed as a DOALL loop.

4.3 Fully Parallel Loop Fusion of Cyclic 2LDGs

In the case of cyclic MLDGs, we begin with the special case where the innermost loop can be DOALL by satisfying the constraints of Theorem 4.2 below. The graph in Figure 2 falls into this category. In the next subsection, we will discuss the general cyclic MLDG.

In order to achieve full parallelism of the innermost fused loop, we need to retime the graph so that the weights of the hard-edges become larger than $(1, -\infty)$, while the weights of the remaining edges are transformed to either be equal to $(0, 0)$ or greater than $(1, -\infty)$. To accomplish this, we retime our 2LDG in two phases. We will demonstrate with the two-dimensional graph of Figure 2.

In the first phase, we retime the first coordinates of each loop dependence vector. We do so by constructing the *constraint graph in x* for G , which we will denote as $G_x^c = (V_x^c, E_x^c, w_x^c)$. As we normally do when constructing a constraint graph, we begin by defining $V_x^c = V \cup \{v_0\}$ and

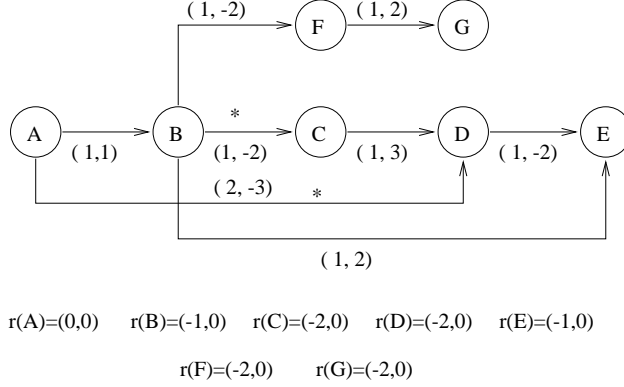


Figure 10: The acyclic 2LDG of Figure 8 after retiming

$E_x^c = E \cup \{v_0 \rightarrow v_i : v_i \in V\}$, where the added edges all have weight zero. We then weight the remaining edges with the first coordinates of the weights in G , i.e. $w_x^c(e) = \delta_L(e)[1]$ for $e \in E$. Finally, we subtract 1 from the weight of each *hard-edge* (previously defined in Section 2.2). Our graph complete, we apply the traditional Bellman-Ford algorithm to find the shortest path lengths from v_0 to every other node. These lengths will be the first coordinates for our retiming function. For example, the constraint graph in x for Figure 2 is pictured in Figure 11(a). Applying the Bellman-Ford Algorithm to it results in initial coordinates for our retiming of $r(A)[1] = r(B)[1] = 0$ and $r(C)[1] = r(D)[1] = -1$. We will later show that the first coordinate of the retimed weight vector for each hard-edge will be zero after this first phase.

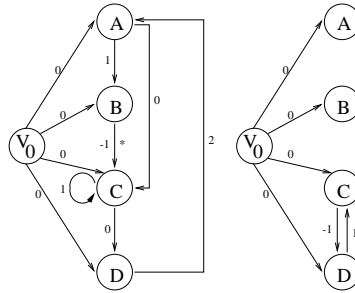


Figure 11: The constraint graphs for Figure 2: (a) in x ; (b) in y

Assuming second coordinates of zero for our retiming for now, we retime the original graph G to start phase two. We then begin constructing the *constraint graph in y* for G , denoted $G_y^c = (V_y^c, E_y^c, w_y^c)$, by including in E_y^c only those edges from E whose edge weights have a zero first coordinate. (Note that, as a result, no hard-edges will be included in this constraint graph.) These

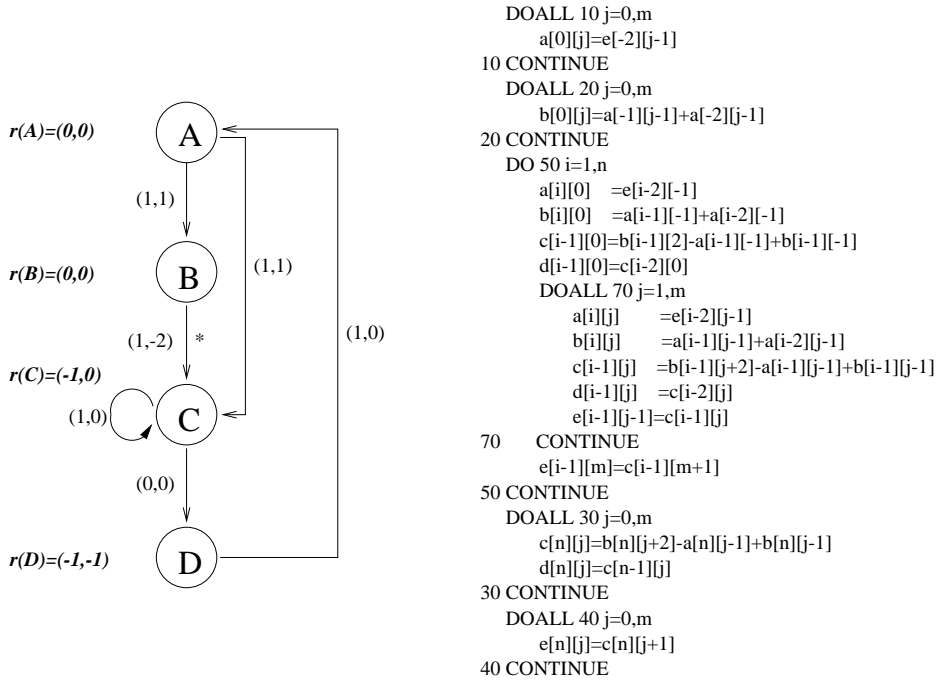


Figure 12: (a) The 2LDG of Figure 2 after retiming; (b) The pseudocode after retiming and fusion

edges will be weighted in the constraint graph by their second coordinates, i.e. $w_y^c(e) = \delta_L(e)[2]$ for all $e \in E_y^c \cap E$. For each of these edges we also add the corresponding back-edge, weighted with the negative of the second coordinates. In other words, if $e : a \rightarrow b$ is an edge in $E \cap E_y^c$, we add an edge from b back to a to the constraint graph with a weight of $-w_y^c(e)$. As usual, V_y^c contains all vertices from V plus an additional v_0 , which is connected to every vertex from V by zero-weight edges. Finally, we execute the Bellman-Ford algorithm on this new constraint graph and find the shortest path lengths, which become the second coordinates for our retiming. Continuing with our example, the constraint graph in y for Figure 2 is given in Figure 11(b). The Bellman-Ford algorithm finds a retiming with $r(A)[2] = r(B)[2] = r(C)[2] = 0$ and $r(D)[2] = -1$.

To summarize, we have found a retiming for the 2LDG of Figure 2 with $r(A) = r(B) = (0, 0)$, $r(C) = (-1, 0)$ and $r(D) = (-1, -1)$. Applying this retiming results in the 2LDG of Figure 12(a) and pseudocode of Figure 12(b). In Figure 13 we show the iteration space after retiming and fusion. It is clear that the innermost loop can be executed in parallel.

We now formally extend our solution for the acyclic graph to cyclic graphs.

Theorem 4.2 *Let $G = (V, E, \delta_L, D_L)$ be a legal 2LDG. Then there is a legal retiming r such that the*

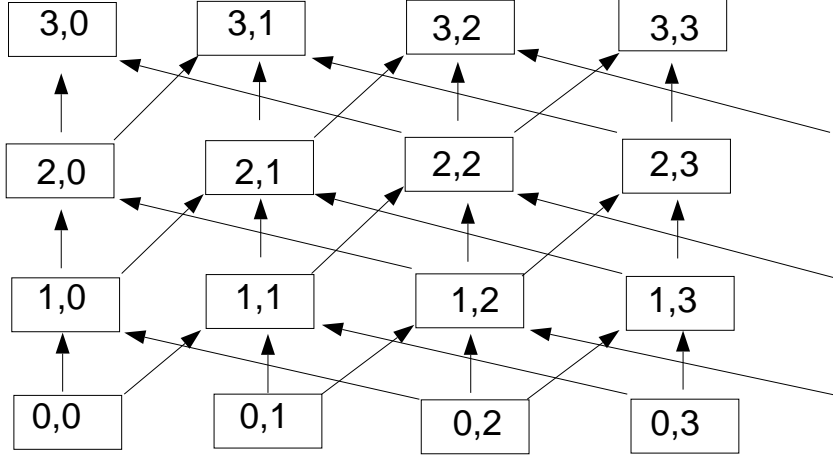


Figure 13: The iteration space of Figure 2 after retiming and fusion

innermost loop nest is DOALL after retiming and loop fusion if and only if the constraint graphs in x and y for G contain no negative-weight cycles.

Proof: Assume that neither constraint graph contains a negative-weight cycle. Let $G - (1, 0) = (V, E, \omega)$ be the 2LDG derived from G by defining $\omega(e) = \delta_L(e) - (1, 0)$ for each hard-edge in E and $\omega(e) = \delta_L(e)$ otherwise. We solve for our retiming in two stages.

- For the first component of the retiming vector, we wish to find a solution for the inequality system

$$r(v_j)[1] - r(v_i)[1] \leq \omega(e)[1] \forall e : v_i \rightarrow v_j \in E.$$

The constraint graph for this system is given by G_x^c . Thus the system has a feasible solution, where $r(v_i)[1]$ is the length of the shortest path from v_0 to v_i in G_x^c . There are now two cases.

1. If $e : v_i \rightarrow v_j$ is a hard-edge, we have $r(v_j)[1] - r(v_i)[1] \leq \delta_L(e)[1] - 1$ by definition.

Thus

$$\delta_{Lr}(e)[1] = \delta_L(e)[1] + r(v_i)[1] - r(v_j)[1] \geq 1,$$

and so $\delta_{Lr}(e) \geq (1, -\infty)$ for all hard-edges e , regardless of what the second component of our retiming turns out to be.

2. If $e : v_i \rightarrow v_j$ is not a hard edge, then $r(v_j)[1] - r(v_i)[1] \leq \delta_L(e)[1]$, and by similar logic $\delta_{Lr}(e)[1] \geq 0$.

- We now wish to find the second component of the retiming vector. Consider only edges $e : v_i \rightarrow v_j$ with $\delta_{Lr}(e)[1] = 0$ which are not hard-edges. (Note that any other edge e' will have $\delta_{Lr}(e') \geq (1, -\infty)$ regardless of what the second component of our retiming is.) For each such edge, add the equation

$$r(v_j)[2] - r(v_i)[2] = \omega(e)[2]$$

to a system of equations. The constraint graph for this system is given by G_y^c , and thus it has a feasible solution. Now, since $\omega(e) = \delta_L(e)$ for all edges $e : v_i \rightarrow v_j$ under consideration at this step,

$$\delta_{Lr}(e)[2] = \delta_L(e)[2] + r(v_i)[2] - r(v_j)[2] = 0,$$

and so each edge e which is not a hard-edge satisfies either $\delta_{Lr}(e) \geq (1, -\infty)$ or $\delta_{Lr}(e) = (0, 0)$.

So for any edge e in G we have one of two cases.

1. If $\delta_{Lr}(e) \geq (1, -\infty)$ then all loop dependence vectors for e satisfy $d_{Lr}(e) \geq (1, -\infty)$.
2. Let $\delta_{Lr}(e) = (0, 0)$ and consider any loop dependence vector $d_{Lr}(e)$. The first coordinate of this vector is a positive integer, and if its larger than 0 we have $d_{Lr}(e) \geq (1, -\infty)$. Suppose then that this first coordinate is 0. Since e is not a hard-edge, all loop dependence vectors for e with first coordinate 0 must have the same second coordinate. Since $\delta_{Lr}(e)$ is one such vector, we conclude that $d_{Lr}(e) = (0, 0)$ in this case.

Thus we have found a retiming function such that, after retiming, all loop dependence vectors satisfy either $d_{Lr}(e) \geq (1, -\infty)$ or $d_{Lr}(e) = (0, 0)$. By Property 4.2, the innermost loop is DOALL after loop fusion.

On the other hand, suppose that one of the constraint graphs for G does contain a negative-weight cycle. If its the constraint graph in x , then the system of inequalities for the first components has no feasible solution, and we cannot find a retiming which insures that the weight of any hard-edge e satisfies $\delta_{Lr}(e) \geq (1, -\infty)$. If the constraint graph in y contains the negative-weight cycle, the system of equations for the second components has no solution, and there may exist an edge e such that $\delta_{Lr}(e) = (0, k)$ for some $k \neq 0$. In either case, by Property 4.2, the innermost loop cannot be DOALL. \square

The algorithm for finding the retiming function for these loops that can be fused with full parallelism according to Theorem 4.2 is given as Algorithm 4 below.

Algorithm 4 Legal Loop Fusion and Full Parallelism for Cyclic 2LDGs

Input: A 2LDG $G = (V, E, \delta_L, D_L)$ **Output:** A retiming function r of the 2LDG or FALSE if none exists.

```
/* — PHASE ONE : Compute first component of retiming function — */
 $V_x^c \leftarrow V \cup \{v_0\}$  /* Construct constraint graph in  $x$ ,  $G_x^c = (V_x^c, E_x^c, w_x^c)$ , from  $G$ . */
 $E_x^c \leftarrow E \cup \{v_0 \rightarrow v_i : v_i \in V\}$ 
for all edges  $e \in E$  do
  if  $e$  is a hard-edge then
     $w_x^c(e) \leftarrow \delta_L(e)[1] - 1$ 
  else
     $w_x^c(e) \leftarrow \delta_L(e)[1]$ 
  end if
end for
for all edges  $e \in E_x^c - E$  do
   $w_x^c(e) \leftarrow 0$ 
end for
 $r_x \leftarrow \text{BellmanFord}(G_x^c)$  /* Use traditional Bellman-Ford algorithm for computation. */
if there is a negative-weight cycle in  $G_x^c$  then
  return FALSE /* The graph  $G$  violates first condition so no solution. */
end if

/* — PHASE TWO : Compute second component of retiming function — */
 $V_y^c \leftarrow V \cup \{v_0\}$  /* Construct constraint graph in  $y$ ,  $G_y^c = (V_y^c, E_y^c, w_y^c)$ . */
 $E_y^c \leftarrow \{v_0 \rightarrow v_i : v_i \in V\}$ 
for all edges  $e \in E_y^c$  do
   $w_y^c(e) \leftarrow 0$ 
end for
for all edges  $e : u \rightarrow v \in E$  do
  if  $e$  is not a hard-edge and  $w_x^c(e) + r_x(u) - r_x(v) = 0$  then
     $E_y^c \leftarrow E_y^c \cup \{e, v \rightarrow u\}$ 
     $w_y^c(e) \leftarrow \delta_L(e)[2]$ 
     $w_y^c(v \rightarrow u) \leftarrow -\delta_L(e)[2]$ 
  end if
end for
 $r_y \leftarrow \text{BellmanFord}(G_y^c)$  /* Again use traditional Bellman-Ford algorithm. */
if there is a negative-weight cycle in  $G_y^c$  then
  return FALSE /* The graph  $G$  violates second condition so no solution. */
end if

/* — PHASE THREE : Combine to get final solution — */
for all nodes  $v \in V$  do
   $r(v) \leftarrow (r_x(v), r_y(v))$ 
end for
return  $r$ 
```

4.4 Fully Parallel Loop Fusion in Hyperplanes for Cyclic 2LDGs

In the case when the constraints established in Theorem 4.2 are not satisfied, the full parallelism of the innermost loop is achieved by using a wavefront approach. We begin showing how to solve this problem by defining a DOALL hyperplane as follows.

Definition 4.1 *A hyperplane is said to be DOALL when all iterations in the hyperplane can be done in parallel.*

The conditions required to obtain a DOALL hyperplane via retiming are expressed in the lemma below:

Lemma 4.3 *Let G be a legal 2LDG transformed by retiming into $G_r = (V, E, \delta_{Lr}, D_{Lr})$. If all retimed dependence vectors $d_{Lr}(e)$ satisfy $d_{Lr}(e) \geq (0, 0)$, then there exists a schedule vector s and a DOALL hyperplane h such that h is perpendicular to s .*

Proof: We need to prove that there exists a schedule vector $s = (s[1], s[2])$ such that $s \cdot d_{Lr}(e) > 0$ for all $d_{Lr}(e) \neq (0, 0)$. We can then let $h = (s[2], -s[1])$ be our choice of hyperplane. Let us assume that the maximum $d_{Lr}(e)$ in lexicographic order is (a, b) where $a \geq 0$. We need to examine two cases:

1. For the case where $a = 0$, all $d_{Lr}(e)$ have the form $(0, k)$ with $k > 0$. Thus $s = (0, 1)$ satisfies the conditions for a DOALL hyperplane.
2. On the other hand, if $a > 0$, we need $s = (s[1], 1)$ such that $s[1] > \frac{-d_{Lr}(e)[2]}{d_{Lr}(e)[1]}$ for all $d_{Lr}(e)$.

Therefore, we choose

$$s[1] = \max \left\{ \left\lfloor \frac{-d_{Lr}(e)[2]}{d_{Lr}(e)[1]} \right\rfloor \right\} + 1$$

and the conditions are satisfied.

□

Finally, the theorem below shows that a retiming function always exist such that a DOALL hyperplane can be found.

Theorem 4.4 *Let $G = (V, E, \delta_L, D_L)$ be a legal 2LDG. If all cycles $c \in G$ satisfy $\delta_L(c) > (0, 0)$, then we can find a legal retiming r such that, after retiming and loop fusion, there is a hyperplane where all iterations are parallel.*

Proof: By Theorem 3.2 we can find a legal retiming function such that, after retiming, all the loop dependence vectors satisfy $d_{Lr}(e) \geq (0, 0)$. By Lemma 4.3 we know that we can find a DOALL hyperplane under these conditions. \square

The algorithm for finding the fully parallel hyperplane is given as Algorithm 5. In order to find the hyperplane, we first calculate the retiming function by the LLOFRA algorithm (Algorithm 2, given in Section 3.2). Next, we calculate the scheduling vector according to Lemma 4.3. Finally, we choose a hyperplane that is perpendicular to the scheduling vector.

Algorithm 5 Full Hyperplane Parallelism for Cyclic 2LDGs

Input: A 2LDG $G = (V, E, \delta_L, D_L)$.

Output: A retiming function r of G and the hyperplane h .

```

 $r \leftarrow$  LLOFRA( $G$ )                                /* Calculate retiming function  $r$  by Algorithm 2. */
 $m \leftarrow (-\infty, -\infty)$                         /* Calculate  $d_{Lr}$  of each edge and find max one. */
for all edges  $e : v_i \rightarrow v_j \in E$  do
  for all  $d_L(e) \in D_L(e)$  do
     $d_{Lr}(e) \leftarrow d_L(e) + r(v_i) - r(v_j)$ 
    if  $m < d_{Lr}(e)$  then
       $m \leftarrow d_{Lr}(e)$ 
    end if
  end for
end for
if  $m[1] = 0$  then
   $s[1] \leftarrow 0$                                   /* Calculate the scheduling vector. */
else
   $s[1] \leftarrow \max \left\{ \left\lfloor \frac{-d_{Lr}(e)[2]}{d_{Lr}(e)[1]} \right\rfloor \right\} + 1$ 
end if
 $s[2] \leftarrow 1$ 
 $h \leftarrow (s[2], -s[1])$                           /* Find the hyperplane. */
return  $r$  and  $h$ 

```

As an example, let's alter the 2LDG of Figure 8 by

- adding edges $D \rightarrow C$ and $E \rightarrow B$;
- defining $D_L(D, C) = \{(0, -2)\}$ and $D_L(E, B) = \{(0, 1), (1, 1)\}$;
- redefining $D_L(C, D) = \{(0, 3), (0, 5)\}$, $D_L(D, E) = \{(0, -2)\}$, and $D_L(A, D) = \{(0, -3), (1, 0)\}$.

This case, shown in Figure 14, can only achieve full parallelism in a hyperplane not parallel to the Cartesian axis. We will use Algorithm 5. We get the retiming function shown in the figure from

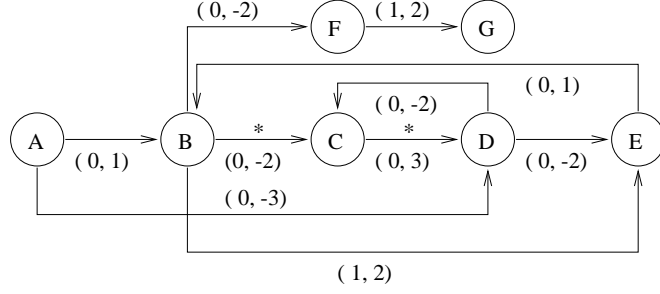
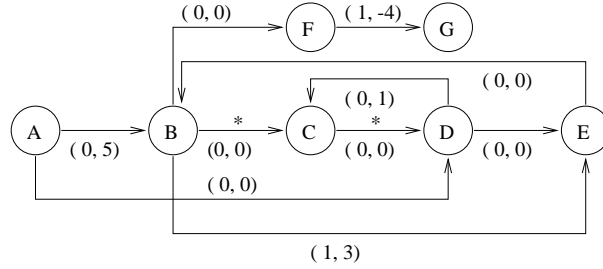


Figure 14: A cyclic 2LDG before retiming



$$\begin{aligned} r(A)=(0,0) \quad r(B)=(0,-4) \quad r(C)=(0,-6) \quad r(D)=(0,-3) \\ r(E)=(0,-5) \quad r(F)=(0,-6) \quad r(G)=(0,0) \end{aligned}$$

Figure 15: The 2LDG of Figure 14 after retiming

Algorithm 2, and then calculate $D_{Lr}(A, B) = \{(0, 5)\}$, $D_{Lr}(B, C) = \{(0, 0), (0, 5)\}$, $D_{Lr}(C, D) = \{(0, 0), (0, 2)\}$, $D_{Lr}(D, C) = \{(0, 1)\}$, $D_{Lr}(D, E) = \{(0, 0)\}$, $D_{Lr}(E, B) = \{(0, 0), (1, 0)\}$, $D_{Lr}(B, F) = \{(0, 0)\}$, $D_{Lr}(F, G) = \{(1, -4)\}$, $D_{Lr}(B, E) = \{(1, 3)\}$, and $D_{Lr}(A, D) = \{(0, 0), (1, 3)\}$. Figure 15 shows the retimed 2LDG. The hardware implementation and the code representing the resulting graph require a detailed description beyond the scope of this paper. However, the interested reader can obtain such details in [22] and [24]. Because the first component of the maximum $d_{Lr}(e)$ is $1 > 0$, we see that the scheduling vector has the form $s = (s[1], 1)$, where $s[1] = \max \left\{ \left\lfloor \frac{-d_{Lr}(e)[2]}{d_{Lr}(e)[1]} \right\rfloor + 1 \right\} = 5$. Thus we get a scheduling vector of $s = (5, 1)$. The hyperplane $h = (1, -5)$ is perpendicular to s , as shown in Figure 16.

5 Experiments

In this section, we demonstrate the effectiveness of our method by experimenting on 5 common MLDGs:

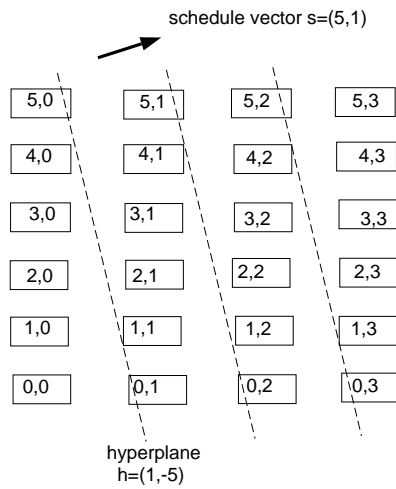


Figure 16: The schedule vector and hyperplane for Figure 14

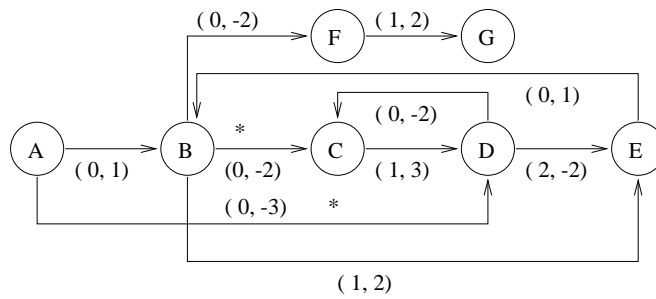
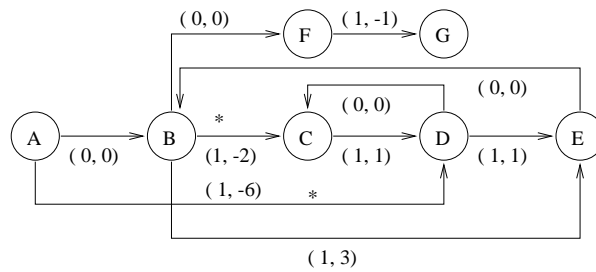


Figure 17: The cyclic 2LDG **Example3** before retiming



$$r(A)=(0,-1) \quad r(B)=(0,0) \quad r(C)=(-1,0) \quad r(D)=(-1,2)$$

$$r(E)=(0,-1) \quad r(F)=(0,-2) \quad r(G)=(0,1)$$

Figure 18: **Example3** after retiming

- Our first three examples are from this paper.
 - **Example1** refers to Figure 8, the acyclic 2LDG from Section 4.2.
 - **Example3** is the variation of Example1 which corresponds to the 2LDG shown in Figure 17. It contains loop-carried dependencies, making the problem more complex. In this case, we notice that the initial fusion can not be done by Algorithm 3 because of the constraints imposed by the existing cycles in the graph. Algorithm 4 is used after we verified that the MLDG satisfies the condition of Theorem 4.2. The resulting retimed MLDG, ready for fusion with fully parallel nested loop, is shown in Figure 18.
 - **Example2** is Figure 14, the cyclic 2LDG of the previous section.
- The fourth experiment, **LL18**, is Kernel 18 of the Livermore Loops. It is an acyclic case with a fusion-preventing dependence, so Algorithm 3 is applied.
- The final two problems are from an astrophysical program for the computation of galactic jets.
 - **JET1** is the first fragment of code from this program. It is acyclic with a fusion-preventing dependence.
 - **JET2** is the second code fragment from this program. There are no fusion-preventing dependences, but there is a hard-edge. Were we to fuse it directly, the result would not be fully parallel. However, after retiming, a legal loop fusion can be done and the full parallelism kept.

Table 1 shows the results for these six problems. The column *alg* shows which algorithm was applied to these examples. The next two columns represent the number of synchronizations before and after fusion, respectively. The column *improvement* measures the improvement after loop fusion. From this table, it is clear that, after our loop fusion algorithms, the number of synchronizations is significantly reduced.

Table 2 shows a theoretical comparison of our results to other loop fusion methods. The column *OPS* shows the characteristics of our method, *Man* represents Manjikian’s method [17], *Carr* represents Carr’s algorithm [7], *Al* is Al-Mouhamed’s approach [3], and *Ken* is Kennedy’s method [13]. The proposed method can perform fusion for all the experiments, and can keep full parallelism after fusion. Only Manjikian can deal with any of these problems, and the results of the four experiments

<i>Program</i>	<i>alg</i>	<i>syn. before</i>	<i>syn. after</i>	<i>improvement</i>
Example1	3	7*n	n-2	700%
Example2	5	7*n	n	700%
Example3	4	7*n	n-1	700%
LL18	3	3*n	n-2	300%
JET1	3	2*n	n-1	200%
JET2	3	2*n	n-1	200%

Table 1: Improvement resulting from applying our algorithms

it can work with are not fully parallel. The other methods cannot do loop fusion for most of the cases, and do not guarantee the parallelism after fusion.

<i>Program</i>	<i>Fusion</i>					<i>Keep full parallelism</i>				
	OPS	Man	Carr	Al	Ken	OPS	Man	Carr	Al	Ken
Example1	Yes	Yes	No	No	No	Yes	Partially	No	No	No
Example2	Yes	No	No	No	No	Yes	No	No	No	No
Example3	Yes	No	No	No	No	Yes	No	No	No	No
LL18	Yes	Yes	No	No	No	Yes	Partially	No	No	No
JET1	Yes	Yes	No	No	No	Yes	Partially	No	No	No
JET2	Yes	Yes	Yes	Yes	No	Yes	Partially	No	No	No

Table 2: Comparison with other loop fusion methods

6 Conclusion

Loop fusion is an effective way to reduce synchronization and improve data locality. Existing loop fusion techniques may not be able to deal with fusion-preventing dependences in nested loops while obtaining a fully parallel fused loop. This paper has introduced a new technique, based on multi-dimensional retiming concepts, which is able to achieve a fully parallel fused loop. Loops are modeled by *loop dependence graphs*, where nodes represent the innermost loops and edges represent data dependences. New algorithms have been presented, covering fusion for trivial acyclic graphs, as well as for complex cyclic ones with fusion-preventing dependences. The theory and preliminary results presented show that it is always possible to efficiently obtain a parallel fused loop by applying this methodology to nested loops.

References

- [1] S. G. Abraham and D. E. Hudak. Compile-time partitioning of iterative parallel loops to reduce cache coherency traffic. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):318–328, July 1991.
- [2] A. Agarwal, D. A. Kranz, and V. Natarajan. Automatic partitioning of parallel loops and data arrays for distributed shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(9):943–962, September 1995.
- [3] M. Al-Mouhamed and L. Bic. Effects of loop fusion and statement migration on the speedup of vector multiprocessors. In *Proc. of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques*, pages 193–202, 1994.
- [4] J. R. Allen and K. Kennedy. Automatic translation of fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.
- [5] U. Banerjee. *Loop Parallelization*. Kluwer Academic Publishers, 1994.
- [6] P.-Y. Calland, A. Darte, and Y. Robert. Circuit retiming applied to decomposed software pipelining. *IEEE Transactions on Parallel and Distributed Systems*, 9(1):24–35, January 1998.
- [7] S. Carr, K. S. McKinley, and C. W. Tseng. Compiler optimizations for improving data locality. In *Proc. of the Sixth International Conference on Architectural Support for Programming Languages and Compilers (ASPLOS-VI)*, pages 252–262, 1994.
- [8] T. H. Corman, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
- [9] A. Darte and Y. Robert. Constructive methods for scheduling uniform loop nests. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):814–822, August 1994.
- [10] P. Feautrier. Some efficient solutions to the affine scheduling problem, part II, multi-dimensional time. *International Journal of Parallel Programming*, 21(6):389–420, December 1992.
- [11] F. H. M. Franssen and F. Balasa. Modeling multidimensional data and control flow. *IEEE Transactions on Very Large Scale Integration Systems*, 1(3):319–326, September 1993.

- [12] C. H. Huang and P. Sadayappan. Communication-free hyperplane partitioning of nested loops. In *Languages and Compilers for Parallel Computing, Springer-Verlag Lecture Notes in Computer Science 589*, pages 186–200. Springer-Verlag, Berlin Germany, 1991.
- [13] K. Kennedy and K. S. McKinley. Optimizing for parallelism and data locality. In *Proc. ACM International Conference on Supercomputing*, pages 323–334, 1992.
- [14] K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Languages and Compilers for Parallel Computing, Springer-Verlag Lecture Notes in Computer Science 768*, pages 301–320. Springer-Verlag, Berlin Germany, 1993.
- [15] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.
- [16] L.-S. Liu, C.-W. Ho, and J.-P. Sheu. On the parallelism of nested for-loops using index shift method. In *Proc. of the 19th International Conference on Parallel Processing*, volume II, pages 119–123, 1990.
- [17] N. Manjikian and T. S. Abdelrahman. Fusion of loops for parallelism and locality. In *Proc. of the 24th International Conference on Parallel Processing*, volume II, pages 19–28, 1995.
- [18] N. Manjikian and T. S. Abdelrahman. Fusion of loops for parallelism and locality. *IEEE Transactions on Parallel and Distributed Systems*, 8(2):193–209, February 1997.
- [19] K. S. McKinley, S. Carr, and C. W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.
- [20] N. L. Passos and E. H.-M. Sha. Full parallelism in uniform nested loops using multi-dimensional retiming. In *Proc. of the 23rd International Conference on Parallel Processing*, volume II, pages 130–133, 1994.
- [21] N. L. Passos and E. H.-M. Sha. Memory/time optimization of 2-d filters. In *Proc. of the IEEE International Conference on Acoustics, Speech & Signal Processing*, volume 5, pages 3223–3226, 1995.
- [22] N. L. Passos and E. H.-M. Sha. Achieving full parallelism using multi-dimensional retiming. *IEEE Transactions on Parallel and Distributed Systems*, 7(11):1150–1163, November 1996.

- [23] N. L. Passos and E. H.-M. Sha. Synchronous circuit optimization via multi-dimensional retiming. *IEEE Transactions on Circuits and Systems, Vol. II - Analog and Signal Processing*, 43(7):507–519, July 1996.
- [24] N. L. Passos and E. H.-M. Sha. Synthesis of multi-dimensional applications in vhdl. In *Proc. International Conference on Computer Design*, pages 530–535, 1996.
- [25] N. L. Passos, E. H.-M. Sha, and S. C. Bass. Loop pipelining for scheduling multi-dimensional systems via rotation. In *Proc. of the 31st Design Automation Conference*, pages 485–490, 1994.
- [26] N. L. Passos, E. H.-M. Sha, and S. C. Bass. Schedule-based multi-dimensional retiming. In *Proc. of 8th International Parallel Processing Symposium*, pages 195–199, 1994.
- [27] A. K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Dept. of Computer Science, Rice University, April 1989.
- [28] J. Ramanujam and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):472–482, October 1991.
- [29] Y. Robert and S. W. Song. Revisiting cycle shrinking. *Parallel Computing*, 18(5):481–496, May 1994.
- [30] J. Warren. A hierachical basis for reordering transformations. In *Proc. 11th ACM Symposium on the Principles of Programming Languages*, pages 272–282, 1984.
- [31] M. Wolf and M. Lam. An algorithmic approach to compound loop transformations. In *Advances in Languages and Compilers for Parallel Computing*, pages 243–259. The MIT Press, Cambridge MA, 1991.
- [32] M. Wolf and M. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.
- [33] M. Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, 1989.
- [34] M. Wolfe. Definition of dependence distance. *ACM Transactions on Programming Languages & Systems*, 16(4):1114–1116, July 1994.

[35] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1996.