Maximize Parallelism for Nested Loops via Loop Striping

Chun Xue[†], Zili Shao[††],Qingfeng Zhuge[†], Meilin Liu[†],Meikang Qiu[†] and Edwin H.-M. Sha[†]

Hong Kong Polytechnic University[††]

University of Texas at Dallas

Summary
The majority of scientific and Digital Signal Processing (DSP) applications are recursive or iterative. Transformation techniques are generally applied to increase parallelism for these nested loops. Most of the existing loop transformations techniques either can not achieve maximum parallelism, or can achieve maximum parallelism but with complicated loop bounds and loop indexes calculations. This paper proposes a new technique, *loop striping*, that can maximize parallelism while maintaining the original row-wise execution sequence with minimum overhead. Loop striping groups iterations into stripes, where a stripe is a group of iterations in which all iterations are independent and can be executed in parallel. Theorems and efficient algorithms are proposed for loop striping transformations. The experimental results show that loop striping always achieves better iteration period than software pipelining and loop unfolding, improving average iteration period by 50% and 54% respectively.

Key words:
Loop Scheduling, Optimization, Loop Transformation.

1. Introduction

Nested loops are the most critical sections in applications such as signal processing, image processing, fluid mechanics, and weather forecasting. To improve the performance on these applications, parallel architectures and systems are generally used. How to generate code for nested loops on parallel architectures is a challenging problem for compilers. This paper proposes a new technique, *loop striping*, that can achieve maximum parallelism while maintaining the original row-wise execution sequence with minimal overhead.

Existing loop transformation methods, like wavefront processing [2] [9], achieve higher level of parallelism for nested loops by changing the execution sequence of the nested loops. This sequence of execution is commonly associated with a schedule vector *s*, also called an ordering vector, which affects the order in which the iterations are performed. The iterations are executed along hyperplanes defined by *s*. When the execution of a hyperplane reaches the boundary of the iteration space, it advances to the next hyperplane according to the direction of *s*. All the iterations on the same hyperplane can be executed in parallel.

Different methods have different means in the selection of an appropriate schedule vector. Among these loop transformation methods, unimodular transformation [6] [14] [15] is one of the major techniques. It unifies loop transformations like loop skewing [16], loop interchange [3], and loop reversal to achieve a particular goal, such as maximizing parallelism or maximizing data locality. The sequence of execution as well as the loop bounds and loop indexes are all changed as the result of unimodular transformation. Another technique, Multi-dimensional retiming [13], restructures the loop body to achieve full parallelism within an iteration. Then the actual scheduling of the fully-parallelized iterations can be done by unimodular transformation [15]. More researches have been conducted on top of unimodular transformation. Anderson and Lam [5] apply unimodular transformation to loop nests to increase the granularity of parallelism. Lim and Lam [4] [11] propose affine transformation that subsumes unimodular transformation to maximize parallelism and minimize synchronization.

Unimodular transformation adds overhead to the transformed loops while achieving higher level of parallelism. First, non-linear index bound checking needs to be conducted on the new loop bounds to assure correctness. Second, loop indexes become more complicated compared to the original loop indexes, and additional instructions are needed to calculate each new index so that the actual array values stored in memory can be correctly referenced.

To have simple loop bounds and simple loop indexes while achieving maximum parallelism, we propose a new loop transformation technique, *loop striping*. Loop striping selects iterations into stripes, where a stripe is a group of iterations in which all the iterations are independent and can be executed in parallel. With proper selection of iterations to be placed into the same stripe, loop striping ensures that all the iterations in the same stripe can be executed in parallel. In this way, it achieves

Manuscript received May 5, 2006.
Manuscript revised May 25, 2006.

higher level of parallelism while maintaining simple loop bounds and loop indexes with minimal overhead. There are two main components in the loop striping technique, loop striping factor and loop striping offset. Loop striping factor determines how many iterations will be included in each stripe, and loop striping offset determines how the iterations will be selected in each stripe. These two components can be tailored to reach the ideal parallelism for any target architecture.

Loop unfolding [12] is another popular transformation technique that can increase parallelism for loops. While both loop striping and loop unfolding group iterations to increase parallelism, loop striping is more advanced than loop unfolding for nested loops. Loop unfolding only unfolds iterations within the same dimension, and it does not change the dependencies between iterations. As a result, there exists a lower bound on iteration period which is the shortest average execution time of an iteration. The best loop unfolding can do is to reach this lower bound. In this paper, we show that loop striping can transform nested loops such that we can always group iterations into stripes, where there is no dependency among iterations in a stripe. Hence, there is no lower bound on iteration period for loop striping. We conduct experiments on a set of digital filters with two dimensional loops. Experimental results show that loop striping always achieves better iteration period than loop unfolding and software pipelining.

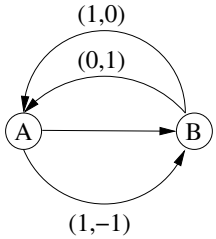
The remainder of this paper is organized as follows. Section 2 provides a motivating example. Section 3 introduces basic concepts and definitions. Theorems and algorithms are proposed in Section 4. Experimental results and concluding remarks are presented in Section 5 and 6, respectively.

2. Motivating Example

In this section, we provide a motivating example to demonstrate the advantage of loop striping compared to loop unfolding and unimodular transformation. The example loop program is shown in Figure 1(a). The MDFG representation of the loop is shown in Figure 1(b). MDFG stands for Multi-dimensional Data Flow Graph. A node in an MDFG represents a computation, and an edge in an MDFG represents a dependence relation between two nodes. Each edge is associated with a delay that helps to identify which two nodes are linked by this edge. For example, in Figure 1(b), the node *A* represent the computation of *A[i, j] = B[i, j-1] + B[i-1, j]* , the edge with delay *(1, -1)* from node *A* to node *B* represents the calculation of *B[i, j]* depends on the value of *A[i-1, j+1]*. The detail formal definitions are presented in section 3.

```
for i=0 to n  do
   for j=0 to m do
      A[i,j] = B[i,j−1] + B[i−1,j];
      B[i,j] = A[i,j] + A[i−1,j+1];
   end for
end for
```

(a)



(b)

Fig. 1: A nested loop and its MDFG.

To show the dependencies among iterations, we often represent iterations in a Cartesian space, called *iteration space*. Figure 2 shows a representation of the iteration space for the MDFG presented in Figure 1(b), in which each node represents an iteration, and each edge represents a dependence relation between two iterations. For example, node *X(1, 1)* represent the iteration of:

$$X(1,1)=\begin{cases} A[1,1]=B[1,0]+B[0,1] \\ B[1,1]=A[1,1]+A[0,2] \end{cases}$$

And there is an edge from node *Y(0, 2)* to node *X(1, 1)* because the calculation of *B[1, 1]* depends on the value of *A[0, 2]*. This can be more easily seen in Figure 3. Figure 3 shows a magnification of the nodes in the iteration space so that we can see the internal operations of each iteration. For simplicity, we will always show a small section of the iteration space with respect to our examples.

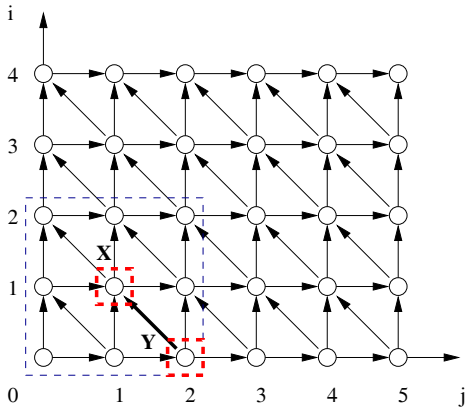


Fig. 2: The iteration space of the original nested loop.

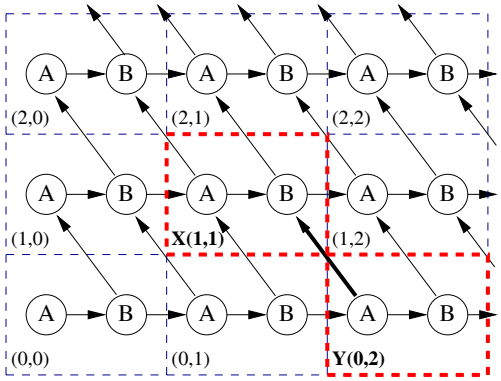
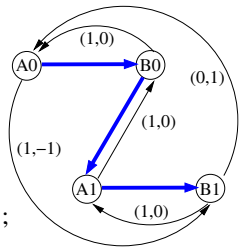


Fig. 3: A close look at the cells of the iteration space.

For the nested loop in Figure 1(a), we apply unfolding with an unfolding factor of two. A new loop program and a new MDFG are obtained as shown in Figure 4. From the new unfolded graph, we can see that the longest path with zero delay is four, which means, all four nodes need to be scheduled sequentially. Figure 5 shows the iteration space after unfolding with an unfolding factor of two. In this example, we do not uncover parallelism by unfolding.

```
for i=0 to n do
  for j=0 to m step by 2 do
     A[i,j] = B[i,j−1] + B[i−1,j];
     B[i,j] = A[i,j] + A[i−1,j+1];
     A[i,j+1] = B[i,j] + B[i−1,j+1];
     B[i,j+1] = A[i,j+1] + A[i−1,j+2];
  end for
end for
```

(a)



(b)

Fig. 4: Loop after unfolding and DFG after loop unfolding.

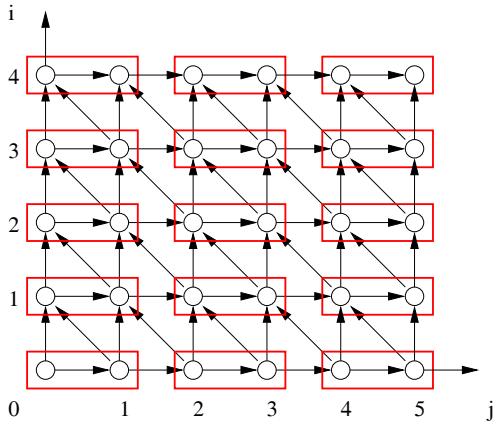


Fig. 5: The iteration space after unfolding.

For the same nested loop in Figure 1, we apply unimodular transformation. The loop becomes:

for *i' = 0* to *2N + M*
  for j' = max(0, ceil((i'- M) / 2)) to min (N, floor(i' / 2))
    A[j',i'-2j'] = B[j', i'-2j'-1] + B[j'-1,i'-2j']
    B[j',i'-2j'] = A[j',i'-2j'] + A[j'-1,i'-2j'+1]
  end for
end for

We can see that both loop bounds and loop indexes calculation become quite complicated. The iteration space of the transformed nest loop is shown in Figure 6.

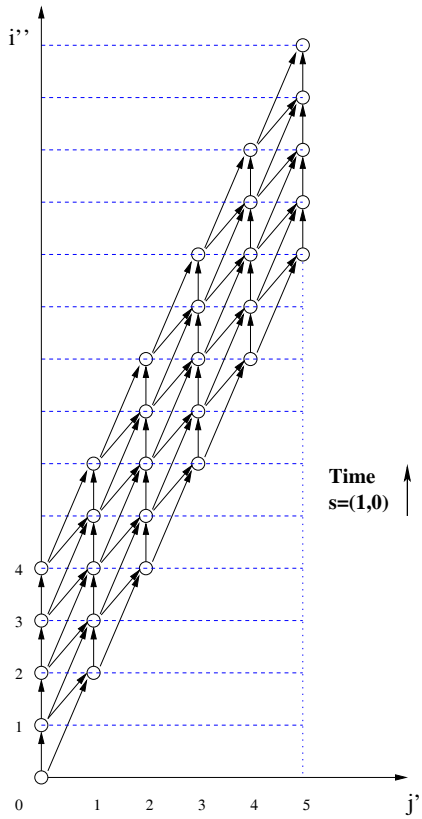


Fig. 6: The iteration space after wavefront transformation.

Figure 7(a) shows the code after loop striping transformation with a striping factor of two and a striping offset of two. Figure 7(b) shows the MDFG after loop striping. Figure 8 shows iterations striped from the original iteration space, and Figure 9 shows the new iteration space with new iteration dependencies. Since the iterations that are striped into the same stripe can be executed in parallel, we reduce the original iteration period by half. Clearly it is better than unfolding with an unfolding factor of two. The code generated by loop striping has simple loop bounds as well as simple loop indexes which unimodular transformation can not achieve. In the rest of this paper, we will present the loop striping technique in detail.

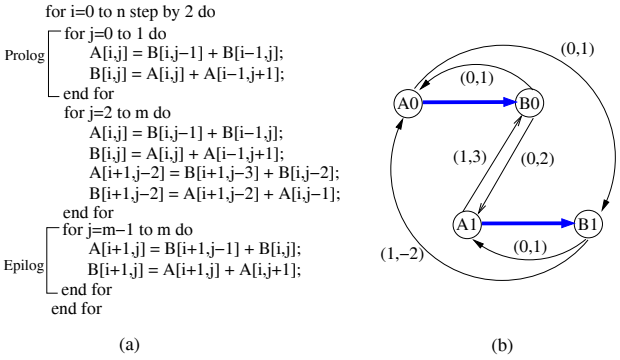


Fig. 7: Loop after striping and its DFG.

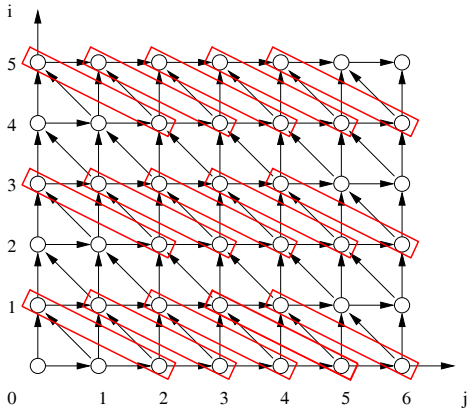


Fig. 8: The iteration space after loop striping.

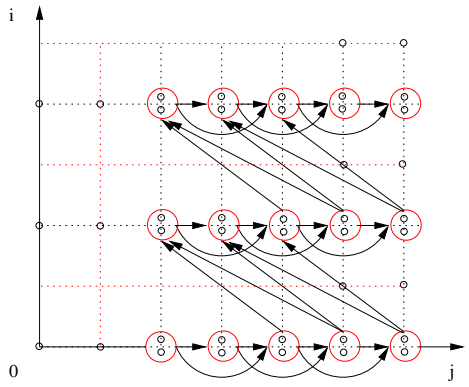


Fig.9: The simplified iteration space after loop striping.

3. Basic Concepts and Definitions

In this section, we introduce the basic concepts which will be used in the later sections. First we present the model and notations that we use to analyze the nested loops.

Second, several related loop transformation techniques are introduced.

Multi-dimensional Data Flow Graph is used to model loops and is defined as follows. A *Multi-dimensional Data Flow Graph (MDFG) G = < V, E, d, t>* is a node-weighted and edge-weighted directed graph, where *V* is the set of computation nodes, *E* is the set of dependence edges, *d* is the multi-dimensional delays between two nodes, also known as dependence vectors, and *t* is the computation time of each node. We use *d(e) = (d.x, d.y)* as a general formulation of any delay shown in a two-dimensional DFG (2DFG). An example is shown in Figure 1. The MDFG in Figure 1(b) models the nested loop in Figure 1(a).

An *iteration* is the execution of each node in *V* exactly once. The computation time of the longest path without delay is called the *iteration period*. For example, the iteration period of the MDFG in Figure 1(b) is 2 from the longest path without delay, which is from node *A* to *B*. Iterations are identified by a vector *i*, equivalent to a MD index. An iteration is associated to a static schedule. A static schedule of a loop is repeatedly executed for the loop. A static schedule must obey the precedence relations defined by the subgraph of an MDFG, consisting of edges without delays. If a node *v* at iteration *j*, depends on a node *u* at iteration *i*, then there is an edge *e* from *u* to *v*, such that *d(e) = j - i*. An edge with delay *(0, 0, ..., 0)* represents a data dependence within the same iteration. A legal MDFG must have no zero-delay cycles.

Iterations can be represented as integral points in a Cartesian space, called *iteration space*, where the coordinates are defined by loop indexes. Such points are identified by a vector *i*, equivalent to a multi-dimensional index. The components of *i* are arranged from the outermost loop control index to the innermost one, always implying a row-wise execution.

A *schedule vector s* is the normal vector for a set of parallel equitemporal hyperplanes that define the sequence of execution of an iteration space. By default, a given nested loop is executed in a row-wise fashion, where the schedule vector *s = (1, 0)*.

To manipulate MDFG characteristics represented on vector notations, such as the delay vectors, we make use of component-wise vector operations. Considering two-dimensional vectors *P* and *Q*, represented by their coordinates *(P.x, P.y)* and *(Q.x, Q.y)*, examples of arithmetic operations are $P + Q = (P.x + Q.x, P.y + Q.y)$ and $P \times Q = (P.x * Q.x, P.y * Q.y)$. The notation $P \cdot Q$ indicates the inner product between *P* and *Q*, i.e., $P \cdot Q = P.x * Q.x + P.y * Q.y$. Vectors are ordered in a left-to-right lexicographic order, i.e., for two n-dimensional vectors $P = (P_1, P_2, P_3, ..., P_n)$ and $Q = (Q_1, Q_2, Q_3, ..., Q_n)$, $P > Q$, if for some $1 \le i \le n$, $P_i > Q_i$ and $\forall\ j < i$, $P_j = Q_j$. For example, *(1, 0, 0) > (0, 2, 1) > (0, 1, 1)*. Vectors are also used to indicate the sequence of computation. In this paper, the sequence of execution is defined as a row-wise computation, i.e., iteration *i* is executed before iteration *j* if $i < j$.

Unfolding is also called unrolling or unwinding. It is widely used in compiler design [1]. A schedule of unfolding factor *f* can be obtained by unfolding the original schedule *f* times. That is, a total of *f* iterations are scheduled together, and the schedule is repeated every *f* iterations. We say the unfolded MDFG $G_f = <V_f, E_f, d_f, t_f>$ is an MDFG obtained by unfolding the original MDFG *f* times. Set V_f is the union of $V^0, V^1, ..., V^{f-1}$. One cycle in G_f consists of all computation nodes in V_f. The period during which all computations in a cycle are executed is called *cycle period*. The cycle period $C(G_f)$ of G_f is defined as:

$$C(G_f) = \max \{ t_f(p_f) \mid d_f(p_f) = 0,\ \forall\ p_f \in G_f \}.$$

Where p_f represent a path in G_f, and $t_f(p_f)$ represent the total computation time of path p_f. During a cycle period of G_f, *f* iterations of *G* are executed. The *iteration period* of G_f is equal to $C(G_f) / f$, in other words, the average computation time for each iteration in *G*. For the original MDFG *G*, the iteration period is equal to *C(G)*. An algorithm can find *C(G)* for an MDFG in time $O(|E|)$ [10].

The *iteration bound* is defined as the maximum time-to-delay ratio of all cycles, $B(G) = \max T(l) / D(l)$ for all cycle $l \in G$ where *T(l)* is the sum of computation time in cycle *l*, and *D(l)* is the sum of delay counts in cycle *l*. A schedule is rate-optimal if the iteration period of this schedule equals its iteration bound. The value *B(G)* can be found in time $O(|V||E|\log|V|)$, when the total number of delays and total computation time are upper bounded by $O(|V|k)$, where *k* is a constant [7]. For a unit-time DFG, it takes only time $O(|V||E|)$ to compute the bound *B(G)* [8].

When there is no resource constraint and a sufficiently large number of iterations are executed together, there is always a static schedule that can achieve the rate-optimality. For a general-time DFG, Parhi and Messerschmitt [12] show that if the unfolding factor is the least common multiple of the delay counts of all cycles, a rate-optimum schedule can be achieved.

Unimodular is a loop transformation technique that unifies all combinations of loop interchange or permutation, skewing and reversal. It can generate an optimal solution in compilation for parallel machines to determine which loop transformations, and in what order, should be applied to achieve a particular goal, such as maximizing parallelism or maximizing data locality [15]. The derivation of the optimal compound transformation consists of two steps. The first step puts the loops into a canonical form, namely a fully permutable loop nest. And the second step transforms the fully permutable loop nest to exploit the target architecture. Specifically, wavefront transformation is used in the second step to maximize the degree of fine-grain parallelism.

4. Loop Striping

In this section, we propose a new loop transformation technique, *loop striping*. First the basic concepts are introduced, and the properties and theorems related to loop striping are discussed. Then the procedures and algorithms to transform the loops after striping are presented. In the following, theorems and algorithms are presented with two dimensional notations, which can be easily extended to multi-dimensions.

4.1 Basic Concepts

In this section, we introduce the theoretical foundations for the proposed loop transformation technique, loop striping.

A stripe is a group of iterations where there is no dependency among the iterations. We call a nested loop after loop striping transformation a striped nested loop. To group iterations into stripes, we need to use loop striping technique defined as follows. Given an MDFG $G = <V, E, d, t>$ representing an n-dimensional nested loop, *loop striping* with vector $s = (f, g)$ will group iterations into stripes. Two important variables for the loop striping technique, f and g, are defined in the following definition.

Striping factor f determines the number of iterations that will be placed into the same stripe. *Striping offset* g is the offset in the inner-most dimension. It determines the direction of the loop striping.

For example, when the striping factor $f = 2$, iteration $(1, 0)$ and iteration $(0, g)$ will be placed in the same stripe. Loop striping groups multiple iterations into one stripe to be scheduled together. With a carefully selected striping offset g, we can group the iterations such that there is no dependency among them. In this way, there is no lower bound on the iteration period. This is the key difference between loop striping and loop unfolding. In the following section, we prove that we can always find such a striping offset g, so that there is no dependency among the striped iterations. Before we prove that we can always find a proper striping offset g, we will first introduce the following lemma.

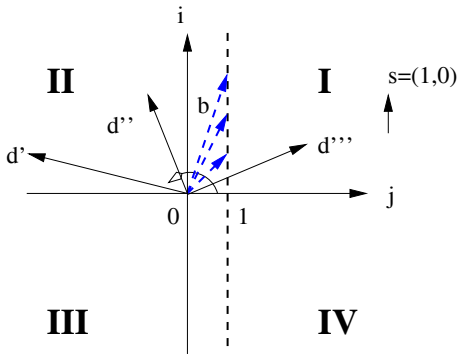


Fig. 10: The relation of vector b and d.

Lemma 4.1. Given an MDFG $G = <V, E, d, t>$ representing an n-dimensional nested loop, and set D is the set that $\forall d \in D, d(e) \neq (0,0)$, we can always find a vector $b = (x, 1)$, such that $\forall d \in D, d \cdot b > 0$.

Proof:

We will prove by finding such a vector $b = (x, 1)$. Since we are given an MDFG that represents an n-dimensional nested loop, by default, this nested loop can be executed in a row-wise fashion.

=> The schedule vector $s = (1, 0)$ is always realizable.

=> $\forall d \in D, d \cdot s \geq 0$.

=> All $d \in D$ stay in region I and II only as shown in Figure 10.

With this understanding, we will find a vector $b = (x, 1)$ as following. We will first sort all $d \in D$ by its angle with j axis. Let d' be such a $d \in D$, that the angle between d' and j axis is the largest. We can easily find a vector $b = (x, 1)$ that can satisfy $d' \cdot b > 0$, this same vector will be able to satisfy $\forall d \in D, d \cdot b > 0$. Here is how we find such a vector $b = (x, 1)$:

Let $d' = (d'_i, d'_j)$, since $b = (x, 1)$, $d' \cdot b > 0$

=> $d'_i \cdot x + d'_j \cdot 1 > 0$

=> $d'_i \cdot x > - d'_j$

=> $x > - d'_j / d'_i$

Since $x > 0$ and x is an integer,

$$x = \begin{cases} -\lceil d'j / d'i \rceil + 1 & d'j < 0 \\ 1 & d'j = 0 \\ 0 & d'j > 0 \end{cases}$$

With this selection of x, we know we can always find such a vector $b = (x, 1)$, that $\forall\, d \in D$, $d \cdot b > 0$.

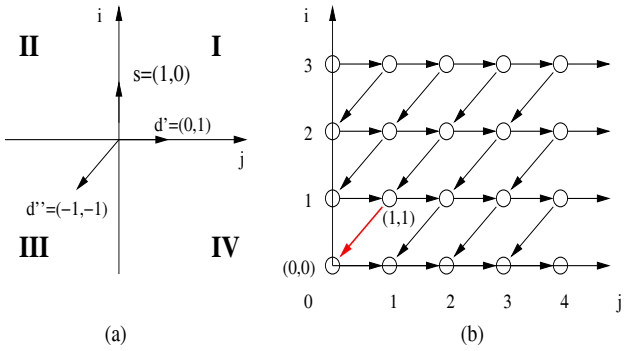


Fig. 11: An example show why $d \cdot s(1,0) \geq 0$

We use an example to explain why when schedule vector is $(1, 0)$, all $d \in D$ will be in region I and II only. As shown in Figure 11(a), we have $d'' = (-1,-1)$ in region III, and $d'' \cdot (1, 0) = -1 < 0$. We can see from the iteration space in Figure 11(b) that with this d'' the schedule is not feasible with the default row-wise scheduling, where schedule vector $s = (1, 0)$ can not be realized. In Figure 11, even the first iteration $(0, 0)$ can not be executed, because it is waiting for the dependency data from iteration $(1, 1)$ that has not completed yet.

Theorem 1. Given an MDFG $G = <V, E, d, t>$ representing an n-dimensional nested loop, a striping offset g can always be found so that there is no dependence between the striped iterations.
Proof:
By definition, assuming a striping factor of 2, for a striping offset g, iteration $(1, 0)$ and iteration $(0, g)$ will be in the same stripe. To prove that we can always find such a g that iteration $(1, 0)$ and iteration $(0, g)$ have no dependency between them, first we describe how to find such a g, and then prove that this g fits our criteria.

Step 1, find g:
Following Lemma 4.1, we can always find a vector $b = (x, 1)$, such that $d(e) \cdot b > 0$ for every $d(e) \neq (0, 0,..., 0)$. We construct a new vector $c = (1, g)$ where $g = x$, so that vector c is orthogonal to b. Use this g as the striping offset.

Step 2, g fits our criteria:
If there is such a delay $d'(e)$ that runs between the iterations in a stripe, then $d'(e)$ is orthogonal to vector b, then $d'(e) \cdot b = 0$. But we know for every $d(e)$, $d(e) \cdot b > 0$. Contradiction.

Therefore, we can always find a striping offset g such that there is no dependence among the striped iterations.

Theorem 2. There is no lower bound on iteration period for nested loops after loop striping transformation is applied.
Proof:
From Theorem 1, we know that we can always find a striping offset g such that there is no dependency among the iterations in a stripe. Then the iteration period can be reduced by dividing the striping factor. For example, if the original iteration period is I, and striping factor is f, then the new iteration period is I / f. With sufficient resources and sufficient number of iterations, we can increase striping factor f to reduce iteration period as much as possible.

After the loop striping transformation, the new program can still keep row-wise execution, which is an advantage over the loop transformation techniques that need to do wavefront execution and need to have extra instructions to calculate complex loop bounds and loop indexes.

4.1 Loop Striping Technique

In this section, we present how to implement the loop striping transformation technique. First, we propose an MDFG transformation algorithm to obtain MDFGs for the striped nested loop. Second, an algorithm to generate the code for loop striping is presented.

Based on a specific architecture, considering the resource constraints, we can find a corresponding loop striping factor and loop striping offset that can achieve the desired parallelism. Then based on the striping factor and striping offset, our algorithms can generate the code and MDFGs for striped nested loops.

4.1.1 The MDFG Transformation Algorithm

The MDFG transformation algorithm for striped nested loops is presented in Algorithm 4.1. The MDFGs for striped nested loops generated from Algorithm 4.1 can help us efficiently perform scheduling for a specific architecture. The properties like iteration period, critical

path, etc., can be easily obtained based on the generated MDFGs. It also provides a foundation for other loop transformation techniques such as Multi-dimensional retiming for further optimization.

Given an MDFG $G = <V, E, d, t>$, loop striping with (f, g) will transform G into $G_s = <V_s, E_s, d_s, t_s>$, where set V_s is the union of $V^0, V^1, ..., V^{f-1}$, and $t_s(u^i) = t(u)$. Each edge e in G is associated with a delay d in the form of $(d(e).i, d(e).j)$. Each edge e_s in G_s is associated with a delay d_s in the form of $(d_s(e_s).i, d_s(e_s).j)$. Procedure to construct E_s and d_s is given as follow.

Algorithm 4.1 The MDFG generation for loop striping

REQUIRE: MDFG $G=<V, E, d, t>$, striping factor f, striping offset g.
ENSURE: E_s and d_s.
for every edge $e = (u, v) \in E$
  $\delta \leftarrow d(e).i \ \% f$;
  $\rho \leftarrow \lceil d(e).i / f \rceil$;
  for $x = 0$ to $f - \delta - 1$
    Add edge $e_s = (u^x, v^{x+\delta})$ to E_s ;
    $d_s(e_s) \leftarrow (\rho, d(e).j + \delta * g)$;
  end for
  for $x = f - \delta$ to $f - 1$
    Add edge $e_s = (u^x, v^{x+\delta-f})$ to E_s ;
    $d_s(e_s) \leftarrow (\rho + 1, d(e).j + (\delta - f) * g)$;
  end for
end for

In this algorithm, for each edge in the original MDFG, we generate f new edges, and assign a proper delay for each new edge. All the dependencies within a stripe are considered intra-stripe dependencies and are represented as edges without delay. The dependency from the stripe (i, j) to the stripe (i', j') is represented by an edge with delay of $(i'-i, j'-j)$. Since f iterations in the i direction compose one stripe, every f delay in $d(e).i$ is represented as 1 delay in $d_s(e_s).i$. The transformation from $d(e).j$ to each copy of $d_s(e_s).j$ is more involved, and the details are given in the algorithm.

Some nice properties about the new MDFG after the loop striping transformation are shown as follows, which can be used for further loop optimization.

PROPERTY 4.1 Let u and v be the nodes in G, $u \xrightarrow{e} v$.

1. The summation of the delays of the f copies of edge e_s is $\sum_{i=0}^{f-1} (d_s(e_s^i)) = (d(e).i , d(e).j * g)$.

2. The f copies of edge e in G_s are the set of edges $u^i \rightarrow v^{(i+d(e).i)\%f}$ for every $0 \le i < f$.

Algorithm 4.1 takes $O(|E|f)$ to execute, where $|E|$ is the number of edges and f is the striping factor.

4.1.2 The Code Transformation Algorithm

We first present some notations. Assume that the original nested loop and the loop striping transformed loop are in the following format:

Original Nested Loop:	Loop Striping transformed Loop:
for $I^1 = L_o^1$ to U_o^1	**for** $I^1 = L_n^1$ to U_n^1 step by S_n^1
for $I^2 = L_o^2$ to U_o^2	**for** $I^2 = L_n^2$ to U_n^2 step by S_n^2
…	…
$B_o(I^1, I^2, I^3, ..., I^i)$	$B_n(I^1, I^2, I^3, ..., I^i)$
..	..
end for	**end for**
end for	**end for**

where $I^1, I^2, I^3, ..., I^i$ are the loop indexes, $L_o^1, L_o^2, L_o^3, …, L_o^i$ are the minimum values for each of the loop indexes in the original loop, $U_o^1, U_o^2, U_o^3, ..., U_o^i$ are the maximum values for each of the loop indexes in the original loop, and $B_o(I^1, I^2, I^3, ..., I^i)$ is the function that represent the loop body of the original loop with $I^1, I^2, I^3, ..., I^i$ as the input parameters. For the striped nested loop, we use the same notations except that subscript n replaces the subscript o.

Using these notations, the algorithm that transform the original nested loop into the new nested loop after striping is given as Algorithm 4.2.

Algorithm 4.2 The code generation for loop striping

REQUIRE: MDFG $G=<V, E, d, t>$, striping factor f, original loop body function $B_o(I^1,I^2,I^3, …, I^i)$, original loop bounds $L_o^1,L_o^2, …, U_o^1,U_o^2,…$
ENSURE: new loop body function $B_n(I^1,I^2, …, I^i)$, new loop bounds $L_n^1,L_n^2, …,U_n^1,U_n^2,…$, new loop steps $S_n^1,S_n^2, …$
  $g \leftarrow$ find_offset(G) (shown in Algorithm 4.3);
  for $x = 0$ to $f-1$
    Append function $B_o(I^1 + x, I^2 - x * g, I^3, ..., I^i)$ to $B_n(I^1, I^2, I^3, ..., I^i)$;
  end for
  for $y = 0$ to i

$L_n^y = L_o^y$;
$U_n^y = U_o^y$;
$S_n^y = 1$;
end for
$L_n^2 = g$;
$S_n^1 = f$;

In the algorithm, we first duplicate the original loop body f times. Each time we increase the loop index I^1 by *1* and decrease the loop index I^2 by striping offset g. After the new loop body is generated, we will change the minimum value of loop index L_n^2 to be g, which means the starting point of the second level loop is offset by the striping offset g. Finally, the step variable of the outer most loops is increased by the striping factor f, which is because we are scheduling f original iterations at a time. In the algorithm, we use the function shown in Algorithm 4.3 to obtain a striping offset.

Algorithm 4.3 Function find_offset(G)

REQUIRE: MDFG $G=< V, E, d, t>$
ENSURE: Striping offset g.
$D \leftarrow \{d \mid d \neq (0, 0, ..., 0)\}$;
Find $d' = (d'_i, d'_j) \in D$ where d'_j / d'_i is the minimum ;

$$g = \begin{cases} -\lceil d'j / d'i \rceil + 1 & d'j < 0 \\ 1 & d'j = 0 \\ 0 & d'j > 0 \end{cases}$$

Return g;

For Algorithm 4.2, it takes $O(|E|)$ time to find the striping offset g, where $|E|$ is the number of edges in the original MDFG. It takes $O(f \times N)$ to complete the code generation, where f is the striping factor and N is the number of instructions in the original loop body. Hence the total time complexity for Algorithm 4.2 is $O(|E| + f \times N)$.

5. Experiment

In this section, we conduct experiments based on a set of DSP benchmarks with two dimensional loops: WDF (Wave Digital Filter), IIR (Infinite Impulse Response Filter), 2D (Two Dimensional Filter), Floyd (Floyd-Steinberg Algorithm), and DPCM (Differential Pulse-Code Modulation Device). Table 1 shows the number of nodes and the number of edges for each benchmark.

Bench	**Nodes**	**Edges**	**Bench**	**Nodes**	Edges
IIR	16	23	WDF	12	16
Floyd	16	20	DPCM	16	23
2D(1)	34	49	MDFG1	4	6
2D(2)	4	6	MDFG2	32	58

Table 1: Benchmarks Information.

For each benchmark, we compare the iteration period of the initial loops, the iteration period of the transformed loop obtained by software pipelining, the iteration period of the transformed loops obtained by loop unfolding, and the iteration period of the transformed loops obtained by loop striping. The results are shown in Table 2. In Table 2, columns "Initial", "S. Pipe.", "Unfolding", and "Striping", represent the iteration periods of the initial loops, the iteration periods after applied software pipelining, the iteration periods of the unfolded loops, and the iteration periods of the striped loops, respectively. Iteration periods in Table 2 are average iteration periods. For unfolded or striped loops, the iteration periods are obtained by two steps: first we calculate the cycle periods for the unfolded or striped loops and then divide the cycle periods by the unfolding factor or striping factor. At the end of Table 2, row "Avg. Iter. Period" shows the average iteration period for each according column. The last row "Iter-re. Avg. Impv." is the average improvement obtained by comparing loop striping with other techniques. Compared to loop unfolding, loop striping reduces iteration period by 54%. Compared to software pipelining, loop striping reduces iteration period by 50%.

From our experiment results, we can clearly see loop striping technique can do much better in uncovering parallelism and increasing timing performance for nested loops than software pipelining and loop unfolding. While software pipelining and loop unfolding improves iteration period for single dimensional loops, loop striping technique significantly reduces iteration period for multi-dimensional loops by exploring multiple dimensions.

Bench	**Iteration Period (cycles)**			
	Unfolding/striping factor=2			
Bench	Initial	S. Pipe.	Unfolding	**Striping**
IIR	5	2	4.5	**2.5**
WDF	6	1	3	**3**

FLOYD	10	8	10	**5**
2D(1)	9	1	5.5	**4.5**
2D(2)	4	4	4	**2**
DPCM	5	2	4.5	**2.5**
MDFG1	7	7	7	**3.5**
MDFG2	10	10	10	**5**
Unfolding/striping factor=4				
Bench	Initial	S. Pipe.	Unfolding	**Striping**
IIR	5	2	3.3	**1.3**
WDF	6	1	1.5	**1.5**
FLOYD	10	8	10	**2.5**
2D(1)	9	1	3.8	**2.3**
2D(2)	4	4	4	**1.0**
DPCM	5	2	3.3	**1.3**
MDFG1	7	7	7	**1.8**
MDFG2	10	10	10	**2.5**
Unfolding/striping factor=6				
Bench	Initial	S. Pipe.	Unfolding	**Striping**
IIR	5	2	2.8	**0.8**
WDF	6	1	1	**1**
FLOYD	10	8	10	**1.7**
2D(1)	9	1	3.2	**1.5**
2D(2)	4	4	4	**0.7**
DPCM	5	2	2.8	**0.8**
MDFG1	7	7	7	**1.2**
MDFG2	10	10	10	**1.7**
Unfolding/striping factor=8				
Bench	Initial	S. Pipe.	Unfolding	**Striping**
IIR	5	2	2.6	**0.6**
WDF	6	1	0.8	**0.8**
FLOYD	10	8	10	**1.3**
2D(1)	9	1	2.9	**1.1**
2D(2)	4	4	4	**0.5**
DPCM	5	2	2.6	**0.6**
MDFG1	7	7	7	**0.9**
MDFG2	10	10	10	**1.3**
Avg. Iter. Period.	7	4.38	4.82	**2.2**
Iter-re. Avg. Impv.	**68%**	**50%**	**54%**	

Table 2: Comparison of iteration period among list scheduling, software pipelining, loop unfolding and loop striping.

6. Conclusion

In this paper, we propose a new loop transformation technique, loop striping. Loop striping can achieve maximum parallelism while maintaining the original schedule vector, namely keeping the row-wise execution sequence. In this way, loop striping simplifies the new loop bounds and loop indexes calculation and reduces overhead.

References

[1] A. Aiken and A. Nicolau. Optimal loop parallelization. ACM Conference on Programming Lauguage Design and Implementation, pages 308-317, 1988.

[2] A. Aiken and A. Nicolau. Fine-Grain Parallelization and the Wavefront Method. MIT Press, 1990.

[3] J. R. Allen and K. Kennedy. Automatic loop interchange. ACM SIGPLAN symposium on Compiler construction, pages 233-246, 1984

[4] G. I. C. Amy W. Lim and M. S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. International Conference on Supercomputing, pages 228-237, 1999.

[5] J. M. Anderson and M. S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. ACM SIGPLAN Conference on Programming Language Design and Implementations, pages 112-125, Jun. 1993.

[6] U. Banerjee. Unimodular Transformations of Double Loops. MIT Press, 1991.

[7] K. Iwano and S. Yeh. An efficient algorithm for optimal loop parallelization. Dec. 1990.

[8] R. M. Karp. A characterization of the minimum cycle mean in a digraph. Discrete Math. 23:309-311, 1978.

[9] L. Lamport. The parallel execution of do loops. Communications of the ACM SIGPLAN, 17:82-93, Feb 1991.

[10] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. Algorithmica, 6:5-35, 1991.

[11] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transformations. ACM SIGPLAN Symposium on Priciple of Programming Languages, pages 201-214, Jan. 1997.

[12] K. K. Parhi and D. G. Messerschmitt. Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding. IEEE Transactions on Computers, 40:178-195, Feb 1991.

[13] N. L. Passos and E. H.-M. Sha. Full parallelism in uniform nested loops using multi-dimensional retiming. International Conference on Parallel Processing, pages 130-133, Aug. 1994.

[14] M.E. Wolf and M. S. Lam. A data locality optimizing algorithm. ACM SIGPLAN conference on Programming Language Design and Implementation, 2:30-44, June 1991.

[15] M.E. Wolf and M. S. lam. A loop transformation theory and an algorithm to maximize parallelism. IEEE Transactions on Parallel and Distributed Systems, 2:452-471, Oct. 1991.

[16] M. Wolfe. Loop skewing: the wavefront method revisited. International Journal of Parallel Programming, 15(4):284-294, Aug. 1986.

Chun Xue received the BS Degree in Computer Science and Engineering from University of Texas at Arlington in May 1997, and MS Degree in computer Science from University of Texas at Dallas, in Dec 2002. He is currently a computer science PhD candidate at University of Texas at Dallas. His research interests include performance and memory optimization for embedded systems, and software/hardware co-design for parallel systems.

Zili Shao received the BE Degree in Electronic Mechanics from University of Electronic Science and Technology of China, China, 1995. He received the MS and PhD Degrees from the Department of Computer Science at the University of Texas at Dallas, in 2003 and 2005, respectively. He has been an Assistant Professor in the Department of Computing at the Hong Kong Polytechnic University since 2005. His research interests include embedded systems, high-level synthesis, compiler optimization, hardware/software co-design and computer security.

Qingfeng Zhuge received her PhD from the Department of Computer Science at the University of Texas at Dallas. She obtained her BS and MS Degrees in Electronics Engineering from Fudan University, Shanghai, China. Her research interests include embedded systems, real-time systems, parallel architectures, optimization algorithms, high-level synthesis, compilers, and scheduling.

Meilin Liu received the BS and MS Degree in Electrical Engineering from Hohai University, Nanjing, China in 1992 and 2000, respectively, and the MS and PhD in computer Science from University of Texas at Dallas, in 2004 and 2006, respectively. Her research interests include optimization of loop execution, loop transformations, and compiler optimization for embedded systems.

Meikang Qiu received BE and ME from Shanghai Jiao Tong University, China; he received MS of Computer Science from University of Texas at Dallas, in 2003 and now he is a Ph.D. candidate there. He has worked at Chinese Helicopter R&D Institute, IBM HSPC, etc. His research interests include embedded systems, information security, etc.

Edwin Sha received the BSE Degree in computer science from National Taiwan University, Taiwan, in 1986, and received MA and PhD Degrees from the Department of Computer Science, Princeton University, in 1991 and 1992, respectively. Since 2000, he has been a tenured full Professor in the Department of Computer Science at the University of Texas at Dallas. He has published more than 200 research papers in refereed conferences and journals. He has been serving as an editor for many journals, and program committee members and chairs in numerous conferences. He received NSF CAREER Award and Teaching award in 1998.