Loop Striping: Maximize Parallelism for Nested Loops *

Chun Xue[1], Zili Shao[2], Meilin Liu[1], Meikang Qiu[1], and Edwin H.-M. Sha[1]

[1] University of Texas at Dallas
Richardson, Texas 75083, USA
{cxx016000, mxl024000, mxq012100, edsha}@utdallas.edu
[2] Hong Kong Polytechnic University
Hung Hom, Kowloon, Hong Kong
cszlshao@comp.polyu.edu.hk

Abstract. 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 transformation 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.

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 and keep the original row-wise execution sequence with minimum overhead.

Existing loop transformation methods, like wavefront processing[2, 8], 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

* This work is partially supported by TI University Program, NSF EIA-0103709, Texas ARP 009741-0028-2001, and NSF CCR-0309461, NSF IIS-0513669, Microsoft, USA.

the iterations are performed. The iterations are executed along hyperplanes defined by s.

Different methods have different means in the selection of an appropriate schedule vector. Among these type of loop transformation methods, unimodular transformation [5, 13, 12] is one of the major techniques. It unifies loop transformations like loop skewing [14], loop interchange [3], and loop reversal to achieve a particular goal, such as maximizing parallelism or 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 [11], 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 [13]. More researches have been building on top of unimodular transformations. Anderson and Lam [4] apply unimodular transforms to loop nests to increase the granularity of parallelism, find the maximum degree of communication-free parallelism across loops, and heuristically introduce communication where necessary.

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 the 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.

While both loop striping and loop unfolding group iterations to increase parallelism, loop striping is more advanced than loop unfolding [10] 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 is a lower bound of iteration period which is the shortest average time to schedule an iteration. Unfolding can only reach this lower bound. We will show that loop striping can transform nested loops in such a way that we can always group iterations into stripes, where there is no dependency between any two iterations in the same 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. The experiment 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 introduces basic concepts and definitions. The theorems and algorithms are proposed in Section 3. Experimental results and concluding remarks are provided in Section 4 and 5, respectively.

2 Basic Concepts and Definitions

In this section, we introduce some basic concepts which will be used in the later sections. First we introduce the model and notion that we use to analyze the nested loops. Second, several related loop transformation techniques are explained.

Multi-dimensional Data Flow Graph is used to model loops and is defined as follows. A *Multi-dimensional Data Flow Graph (MDFG)* $G = \langle V, E, d, t\rangle$ is a node-weighted and edge-weighted directed graph, where V is the set of computation nodes, $E \subseteq V * V$ 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 *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.* 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 are represented as integral points in a Cartesian space, called *iteration space* , where the coordinates are defined by the loop control indexes. Such points are identified by a vector $\widehat{i}$, equivalent to a multi-dimensional index. The components of $\widehat{i}$ 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 a sequence of execution of an iteration space. By default, a given nested loop is executed in a row-wise fashion, where the schedule vector $s = (1, 0)$.

Unfolding is also called unrolling or unwinding, is widely used in compiler design [1]. A schedule of unfolding factor f can be obtained by unfolding G 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 a MDFG obtained by unfolding G 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 equals max$\{t_f(p_f) \mid d_f(p_f)$=0 $\forall$ p_f in $G_f\}$. During a cycle period of G_f, f iterations of G are executed. Thus, 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 a MDFG in time $O(|E|)$[9].

The *iteration bound* is defined to be 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[6]. For a unit-time DFG, it takes only time $O(|V||E|)$ to compute the bound $B(G)$ [7].

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 [10] showed 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 unify all combinations of loop interchange or permutation, skewing and reversal. It can generate an optimal solution in compilation for parallel machines that which loop transformations, and in what order, should be applied to achieve a particular goal, such as maximizing parallelism or data locality [13]. 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 then transforms the fully permutable loop nest to exploit according to the target architecture. Specifically, to maximize the degree of fine-grain parallelism, wavefront transformation is used in the second step.

3 Loop Striping

In this section, we propose a new loop transformation technique, *loop striping.* First the basic concepts are introduced, and the property and theorems related to loop striping are discussed. Then the procedures and algorithm 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.

3.1 Basic Concepts

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

Definition 1. *A stripe is a group of iterations that there is no dependency between any two of the iterations.*

We call a nested loop after loop striping transformation as 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 2. *Striping factor f determine the number of iterations that will be placed into the same stripe. Striping offset g determine the direction of the loop striping, where iteration (1,0) and iteration (0,g) will be placed in the same stripe if the striping factor is 2.*

Loop Striping groups multiple iterations into one stripe to be scheduled together. With a carefully selected striping offset g, we can group the iterations where there are no dependency among them. In this way, there is no lower bound on the iteration period given no resource constraint and sufficiently large number of iterations. This is the key difference between loop striping and loop unfolding. In the following section, we will 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.

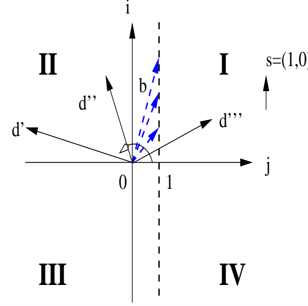

Fig. 1. The relation of vector b and d ∈ D.

Lemma 1. *Given a 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 a MDFG that represents an n-dimensional nested loop, by default, this nested loop can be executed in a row-wise fashion.
$\Rightarrow$ The schedule vector s=(1,0) is always realizable.
$\Rightarrow$ $\forall$ d $\in$ D, d $\cdot$ s $\geq$ 0.
$\Rightarrow$ All d $\in$ D stay in region I and II only as shown in Figure 1.

With this understanding, we will find the 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, such 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 b=(x,1):

Let d'=(d'_i, d'_j), since b=(x,1) and d' $\cdot$ b $>$ 0
$\Rightarrow$ $d'_i \cdot x + d'_j \cdot 1 > 0 \Rightarrow d'_i \cdot x > -d'_j \Rightarrow x > -d'_j/d'_i$

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

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 an 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 g fits our criteria.

Step 1, find g:

Following Lemma 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. then this g is 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 between the striped iterations.

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 loop bounds and loop indexes.

3.2 The Loop Striping Technique

In this section, we present how to implement the loop striping transformation technique. 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.

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
...	...
$B_o(I^1, I^2, I^3, \ldots, I^i)$	$B_n(I^1, I^2, I^3, \ldots, I^i)$
...	...
end for	end for
end for	end for

where $I^1, I^2, I^3, \ldots, I^i$ are the loop indexes, $L_o^1, L_o^2, L_o^3, \ldots, L_o^i$ are the minimum values for each of the loop indexes in the original loop, $U_o^1, U_o^2, U_o^3, \ldots, U_o^i$ are the maximum values for each of the loop indexes in the original loop, and $B_o(I^1, I^2, I^3, \ldots, I^i)$ is the function that represent the loop body of the original loop with $I^1, I^2, I^3, \ldots, I^i$ 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 3.1.

Algorithm 3.1 The code generation for loop striping

Require: DFG $G = \langle V, E, d, t \rangle$, the striping factor f, the original loop body function $B_o(I^1, I^2, I^3, \ldots, I^i)$, the original loop bounds $L_o^1, L_o^2, \ldots, U_o^1, U_o^2, \ldots$

Ensure: the new loop body function $B_n(I^1, I^2, \ldots, I^i)$, the new loop bounds $L_n^1, L_n^2, \ldots, U_n^1, U_n^2, \ldots$, the new loop steps $S_n^1, S_n^2, \ldots$

$g \leftarrow$ find_offset(G) (shown in Algorithm 3.2);
for x = 0 to f-1 do **do**
  Append function $B_o(I^1 + x, I^2 - x * g, I^3, \ldots, I^i)$ to $B_n(I^1, I^2, I^3, \ldots, I^i)$;
end for
for y = 0 to i do **do**
  $L_n^y = L_o^y$; $U_n^y = U_o^y$; $S_n^y = 1$;
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 loop 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 3.2 to obtain a striping offset. This function follows the steps described in the proof of Lemma 1.

For algorithm 3.1, 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

Algorithm 3.2 Function find_offset(G)

Require: MDFG $G = \langle V, E, d, t \rangle$
Ensure: Striping offset g
  D $\leftarrow \{d | d \neq (0, 0, \ldots 0)\}$;
  Find d'=$(d'_i, d'_j) \in$ D that 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;

of instructions in the original loop body. Hence the total time complexity for Algorithm 3.1 is $O(|E| + f \times N)$.

4 Experiments

In this section, we conduct experiments based on a set of DSP benchmarks with two dimensional loops: WDF (Wave Digital Filter), IIR (the Infinite Impulse Response Filter), 2D (the Two Dimensional filter), Floyd (Floyd-Steinberg algorithm), and DPCM (Differential Pulse-Code Modulation device).

For each benchmark, we compare the iteration periods of the initial loops, the iteration periods for the transformed loops obtained by software pipelining, the iteration periods for the transformed loops obtained by loop unfolding, and the iteration periods of the transformed loops obtained by loop striping. The results are shown in Table 1. In the Table1, 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 the table are average iteration periods. For unfolded or striped loops, the iteration periods are obtained by two steps: first calculate the cycle periods for unfolded or striped loops and then divide the cycle periods by the unfolding factor or striping factor. At the end of Table 1, 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, iterational retiming reduces iteration period by 54%. Compared to software pipelining, iterational retiming reduces iteration period by 50%.

From our experiment results, we can clearly see loop striping technique can do much better in increasing parallelism and timing performance on 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 further for multi-dimensional loops. As the unfolding/striping factor grows larger, the improvement becomes more and more substantial. Having a smaller iteration period means that we

Benchmark	**Iteration Period (cycles)**			
unfolding/striping factor=2				
Benchmark	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				
Benchmark	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				
Benchmark	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				
Benchmark	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 1. Comparison of iteration period among list scheduling, software pipelining, loop unfolding and loop striping.

can complete each iteration faster. As a result, our technique can significantly improve the timing performance for applications with nested loops.

5 Conclusion

In this paper, we propose a new loop transformation technique, loop striping. Loop striping can achieve the 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. We believe loop striping is a promising technique and can be applied to different fields for nested loop optimization.

References

1. A. Aiken and A. Nicolau. Optimal loop parallelization. *ACM Conference on Programming Language 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. 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.
5. U. Banerjee. *Unimodular Transformations of Double Loops.* MIT Press, 1991.
6. K. Iwano and S. Yeh. An efficient algorithm for optimal loop parallelization. Dec. 1990.
7. R. M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics*, 23:309–311, 1978.
8. L. Lamport. The parallel execution of do loops. *Communications of the ACM SIG-PLAN*, 17:82–93, FEB. 1991.
9. C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, 1991.
10. 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.
11. 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.
12. 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.
13. 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.
14. M. Wolfe. Loop skewing: the wavefront method revisited. *International Journal of Parallel Programming*, 15(4):284–294, Aug. 1986.