Iterational Retiming: Maximize Iteration-Level Parallelism for Nested Loops *

Chun Xue, Zili Shao, Meilin Liu, Edwin H.-M. Sha
Department of Computer Science
University of Texas at Dallas
{cxx016000,zxs105000,mxl024100,edsha}@utdallas.edu

ABSTRACT

Nested loops are the most critical sections in many scientific and Digital Signal Processing (DSP) applications. It is important to study effective and efficient transformation techniques to increase parallelism for nested loops. In this paper, we propose a novel technique, iterational retiming, that can satisfy any given timing constraint by achieving full parallelism for iterations in a partition. Theorems and efficient algorithms are proposed for iterational retiming. The experimental results show that iterational retiming is a promising technique for parallel embedded systems. It can achieve 87% improvement over software pipelining and 88% improvement over loop unfolding on average.

Categories and Subject Descriptors

D.3.4 [**Software Engineering**]: Processors—*Optimization*

General Terms

Languages, Performance, Algorithms

Keywords

Retiming, Partition, Nested Loops, Optimization

1. INTRODUCTION

Design of Application Specific Integrated Circuits (ASIC) is usually required in order to improve the execution performance of computation-intensive applications. A large group of such applications are multi-dimensional problems, i.e., problems involving more than one dimension, such as computer vision, high-definition television, medical imaging, and remote sensing. Nested loops are the most critical sections in these applications. With parallel application specific architectures being widely used to process these applications, it is important to study effective and efficient transformation techniques to increase parallelism for nested loops. This paper answers a question for parallel embedded systems: Given a timing constraint, can a schedule be generated to satisfy this timing constraint, and if yes, how should the schedule be generated. A new technique, iterational retiming, is proposed that can achieve any timing requirement for nested loops with minimum overhead.

There has been a lot of research on program transformations to enhance parallelism for loops. Numerous techniques have been proposed for one-dimensional loops [1, 3, 4, 6]. Renfors and Neuvo have proved that there is a lower bound of iteration period for one-dimensional loops [8]. Optimal scheduling for one-dimensional loops could reach this lower bound[4], but can not do better than this lower bound. In this paper, we show that there is no lower bound of iteration period for multi-dimensional loops. Using our proposed technique, we can achieve any timing requirement.

A lot of works also have been done for nested loops to increase parallelism. Majority of these works are based on wavefront transformation [2, 5], which achieve higher level of parallelism for nested loops by changing the execution sequence of the nested loops. Wavefront 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 higher level of parallelism, we propose a new loop transformation technique, iterational retiming. Iterational retiming first partitions the iteration space into basic partitions, and then retiming is performed at iteration level so that all the iterations in each partition can be executed in parallel. In this way, we achieve higher level of parallelism while maintaining simple loop bounds and loop indexes with virtually no overhead. Retiming [6, 7] has been widely applied to increase instruction level parallelism. We apply the retiming technique to iterations instead of instructions, and we show that full-parallelism for iterations in each partition can always be achieved by iterational retiming. Experimental results show that our proposed technique can always reach the given timing requirement, while techniques like

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

CODES+ISSS'05, Sept. 19–21, 2005, Jersey City, New Jersey, USA.

loop unfolding, and software pipelining can not always do so.

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 notions that we use to analyze nested loops. Second, loop partitioning technique is introduced. In this paper, our technique is presented with two dimensional notations. It can be easily extended to multi-dimensions.

2.1 Modeling Nested Loops

Multi-dimensional Data Flow Graph is used to model nested 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 a function and d(e) is the multi-dimensional delays for each edge $e \in E$ which is also known as dependence vector, 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.

```
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
```

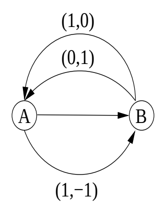

Figure 1: A nested loop and its MDFG.

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 is 2 from the longest path, which is from node A to B. 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) represents a data dependence within the same iteration. A legal MDFG must not have 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 $\hat{i}$, equivalent to a multi-dimensional index. The components of $\hat{i}$ are arranged from the outermost loop control index to the innermost loop control index, which always implies 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 $s = (1, 0)$.

Retiming [6] can be used to optimize the cycle period of a DFG by evenly distributing the delays in it. Given a MDFG $G = \langle V, E, d, t \rangle$, retiming r of G is a function from V to integers. For a node $u \in V$, the value of $r(u)$ is the number of delays drawn from each of its incoming edges of node u and pushed to all of its outgoing edges. Let $G_r = \langle V, E_r, d_r, t \rangle$ denote the retimed graph of G with retiming r, then $d_r(e) = d(e) + r(u) - r(v)$ for every edge $e(u \to v) \in E_r$ in G_r.

2.2 Partitioning the Iteration Space

Instead of executing the entire iteration space in the order of rows and columns, we can first partition it and then execute the partitions one by one. The two boundaries of a partition are called the *partition vectors*. We will denote them by P_x and P_y. Due to the dependencies in the MDFG, partition vectors cannot be arbitrarily chosen. For example, consider the iteration space in Figure 2. Iterations are now represented by dots and inter-iteration dependencies. Partitioning the iteration space into rectangles, with $P_x = (0, 1)$ and $P_y = (2, 0)$, as shown in Figure 2(a), is illegal because of the forward dependencies from $Partition_{(0,0)}$ to $Partition_{(0,1)}$ and the dependencies from $Partition_{(0,1)}$ to $Partition_{(0,0)}$. Due to these two-way dependencies between partitions, we cannot execute either one first. This partition is therefore not implementable and is thus illegal. In contrast, consider the alternative partition shown in Figure 2(c), with $P_x = (0, 1)$ and $P_y = (2, -2)$. Since there is no two-way dependency, a feasible partition execution sequence exists. For example, execute $Partition_{(0,0)}$ first, then $Partition_{(0,1)}$, and so on. Therefore, it is a legal partition.

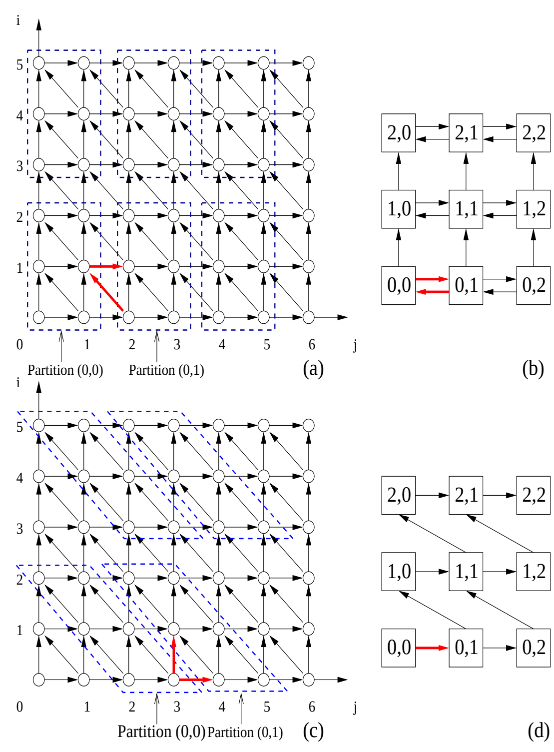

Figure 2: (a) An illegal partition of the iteration space (b) Partition dependency graph (c) A legal partition (d) Partition dependency graph

Iteration Flow Graph is used to model nested loop partitions and is defined as follows. An *Iteration Flow Graph*

(IFG) $G_i = \langle V_i, E_i, d_i, t_i\rangle$ is a node-weighted and edge-weighted directed graph, where V_i is the set of iterations in a partition. The number of nodes $|V_i|$ in an IFG G_i is equal to the number of nodes in a partition. $E_i \subseteq V_i * V_i$ is the set of iteration dependence edges. d_i is a function and $d_i(e)$ is the multi-dimensional delays for each edge $e \in E_i$. t_i is the computation time for each iteration. An iteration flow graph $G_i = \langle V_i, E_i, d_i, t_i\rangle$ is *realizable* if the represented partition is legal. An example of IFG for the basic partition in Figure 2(c) is shown in Figure3. Edges with (0,0) delay are shown in thicker line.

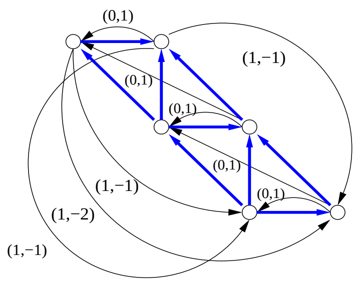

Figure 3: IFG for the partition in Figure2(c)

3. ITERATIONAL RETIMING

In this section, we propose a new loop transformation technique, iterational retiming. First the basic concepts and the theorems related to iterational retiming are discussed. Then the procedures and algorithms to transform the loops are presented in the second section.

3.1 Definitions and Theorems

Iterational retiming is carried out in the following steps. Given a MDFG, first the directions of legal partition vectors will be decided. Second, partition size will be determined to meet the input timing constraint. Third, iterational retiming will be applied to create the retimed partition.

Among all the delay vectors in a MDFG, two extreme vectors, the clockwise (CW) and the counterclockwise (CCW), are the most important vectors for deciding the directions of the legal partition vectors. Legal partition vector cannot lie between CW and CCW. In other words, they can only be outside of CW and CCW or be aligned with CW or CCW. For example, for the nested loop in Figure 1, there are three non-zero delay vectors, (0,1),(1,0) and (1,-1). (0,1) is the CW vector and (1,-1) is the CCW vector. If partition vectors are chosen to be $P_x = (0,1)$ and $P_y = (2,0)$, as shown in Figure 2(a), it is an illegal partition because partition vector (2,0) lies between CW and CCW and causes cycles in the partition dependency graph as shown in Figure 2(b). For the basic partition in our algorithm, we choose P_x to be aligned with x-axis, and P_y to be aligned with CCW. This is a legal choice of partition vectors because the y elements of the delay vectors of the input MDFG are always positive or zero, which allows the default row-wise execution of nested loops. For convenience, we use P_{x0} and P_{y0} to denote the base partition vectors. The actual partition vectors are then denoted by $P_x = f_x P_{x0}$ and $P_y = f_y P_{y0}$, where f_x and f_y are called partition factors, which are related to the size of the partition.

After basic partition is identified via P_x, and P_y, an IFG $G_i = \langle V_i,\ E_i,\ d_i,\ t_i\rangle$ can be constructed. An *iterational retiming* r is a function from V_i to Z^n that redistributes the iterations in a partition. A new IFG $G_{i,r}$ is created, such that the number of iterations included in the partition is still the same. The retiming vector r(u) of an iteration $u \in G_i$ represents the offset between the original partition containing u, and the one after iterational retiming. When all the edges $e \in E_i$ have non-zero delays, all the nodes $v \in V_i$ can be executed in parallel, which means all the iterations in a partition can be executed in parallel. We call such a partition a *retimed partition*. Properties, algorithms and supporting theorems for iterational retiming are presented below. We will first show how to choose f_x so that retimed partition can be achieved.

Given a MDFG G, let min_k be the minimum k of all the (0, k) delays in G. Let f_x be the size of partition in the x dimension. Given an Iteration Flow Graph (IFG) after we partition the iteration space with f_x and f_y, we want to make sure that the IFG can be retimed to be fully parallel using basic retiming $r = (0, 1)$. There are two types of cycles in an IFG, one with delay $d(c) = (0, y)$ and the other with delay $d(c) \geq (1, -\infty)$. The cycles with delays $d(c) \geq (1, -\infty)$ can be easily retimed to be fully parallel by using $r = (0, 1)$. But for cycles with delays $d(c) = (0, y)$, y must be $\geq$ n(c), where n(c) denotes the number of nodes in cycle c, in order to distribute (0, 1) delay to each edge in cycle c. To simplify notations, we just focus on the $(0, k)$ cycles and delays, so when we say $d(c) \geq n(c)$, it means that $d(c) = (0, y)$, $y \geq n(c)$.

Property 1. Given f_x, an edge with d(e) = (0, b) in DFG will become f_x edges in IFG from iteration node i (here we use i to denote (0, i)), $0 \leq i < f_x$, to node (i+b) mod f_x with delay = (i +b) div f_x.

THEOREM 1. *If $f_x > min_k$, in the resulting Iteration Flow Graph (IFG), there exists a cycle c where $d(c) < n(c)$ and $n(c)$ denotes the number of nodes in cycle c.*

Proof: *Case 1: min_k is relatively prime to f_x. Based on the group theory, the min_k is a generator for the group (mod f_x, "+"). The sequence starting from iteration node 0, to iteration (0+min_k) div f_x, to (0+2 min_k) div f_x, ..., etc. must form a cycle. It is obvious the first edge has 0 delay and each delay along this cycle must be $\leq (0, 1)$. Thus $d(c) < n(c)$. For example, min_k = 3, f_x=4, the cycle from node 0:(0, 3, 2, 1,0). d((0, 3)) = (0, 0).*
Case 2: gcd (min_k, f_x) = a > 1. From the group theory, there must form a cycle from node 0 with f_x/a edges. It is obvious the delay of the first edge is 0 and others $\leq (0, 1)$ along c. Thus, $d(c) < n(c)$. For example, min_k = 2, f_x=6, the cycle from node 0: (0, 2, 4, 0), d((0, 2))= (0, 0).

THEOREM 2. *If $f_x \leq min_k$, for each cycle c in IFG, d(c) must be $\geq n(c)$.*

Proof: *Considering a delay (0, b) in DFG, whose b must be $\geq f_x$ from the condition of the theorem.*
Case 1: b = (0, z × min_k), $z \geq 1$. This will cause self cycles in IFG with (0, z) delay, $z \geq 1$. So $d(c) \geq n(c)$.
Case 2: $b > f_x$. Based on the property described previously, for a node i in IFG, i → (i +b) mod f_x has a delay (i+b) div f_x. It is obvious that $(i + b) > f_x$, so the delay $\geq (0, 1)$. Thus, $d(c) \geq n(c)$ for each c.

As a result of the above theorems, we know that let $f_x \le min_k$, and $r = (0, 1)$ as the retiming function, a basic partition can be retimed into retimed partition. After the iterational retiming 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 Iterational Retiming Technique

In this section, we will present how we can implement the iterational retiming transformation technique. First, we propose a MDFG transformation algorithm to obtain IFG from a basic partition. Second, algorithm to perform iterational retiming is presented.

3.2.1 The MDFG Transformation Algorithm

The MDFG to IFG transformation algorithm is presented in Algorithm 3.1. With IFG, we can perform iterational retiming to achieve full-parallelism at the iteration level for each partition.

Given a MDFG $G = \langle V, E, d, t\rangle$ and basic partition with $P_x = (0, f_x)$, $P_y = (g, f_y)$, we will transform G into $G_i = \langle V_i, E_i, d_i, t_i\rangle$, where V_i is the set of $\{(v^{(0,0)}, v^{(0,1)}, ..., v^{(0,f_x-1)}), (v^{(1,0)}, v^{(1,1)}, ..., v^{(1,f_x-1)}), ... , (v^{(f_y-1,0)}, v^{(f_y-1,1)}, ..., v^{(f_y-1,f_x-1)})\}$
Each edge e in G is associated with a delay d in the form of $(d(e).i, d(e).j)$. Each edge e_i in G_i is associated with a delay d_i in the form of $(d_i(e_i).i, d_i(e_i).j)$. Procedure to construct E_i and d_i is presented as follow.

Algorithm 3.1 IFG generation

Require: MDFG $G = \langle V, E, d, t\rangle$, partition vectors $P_x = (0, f_x)$, $P_y = (g, f_y)$
Ensure: E_i and d_i

```
for all every edge e=(u,v) in E do
  δ ← d(e).i % f_y; ρ ← ⌊d(e).i / f_y⌋ ;
  α ← (d(e).j + d(e).i × g/f_y) % f_x;
  β ← ⌊(d(e).j + d(e).i × g/f_y) / f_x⌋;
  for m = 0 to f_y-δ-1 do
    for n = 0 to f_x-α-1 do
      Add edge e_i=(v^(m,n), v^(m+δ,n+α)) to E_i ;
      d_i(e_i) ← (ρ, β );
    end for
    for n = f_x-α to f_x-1 do
      Add edge e_i=(v^(m,n), v^(m+δ,n+α−f_x)) to E_i ;
      d_i(e_i) ← (ρ, β + 1 );
    end for
  end for
  for m = f_y-δ to f_y-1 do
    for n = 0 to f_x-α-1 do
      Add edge e_i=(v^(m,n), v^(m+δ−f_y,n+α)) to E_i ;
      d_i(e_i) ← (ρ + 1, β );
    end for
    for n = f_x-α to f_x-1 do
      Add edge e_i=(v^(m,n), v^(m+δ−f_y,n+α−f_x)) to E_i ;
      d_i(e_i) ← (ρ + 1, β + 1 );
    end for
  end for
end for
```

In this algorithm, for each edge in the original MDFG, we generate $f_x \cdot f_y$ new edges, and assign a proper delay for each new edge. All the dependencies within a partition are considered intra-partition dependencies and are represented as edges with zero delays. Since f_y rows iterations in the i direction are inside one partition, every f delay in $d(e).i$ is represented as 1 delay in $d_i(e_i).i$. The transformation from $d(e).j$ to each copy of $d_i(e_i).j$ is more involved, and the details are given in the algorithm.

3.2.2 The Iterational Retiming Algorithm

The iterational retiming algorithm is first presented and explained. An example is given at the end of this section.

Algorithm 3.2 Iterational Retiming

Require: MDFG $G = \langle V, E, d, t\rangle$, timing requirement T
Ensure: A retimed partition that meets timing requirement

```
/* Step 1. Based on the input MDFG, find a basic partition that is legal and have enough number of iterations to meet the timing requirement T; */
c ← cycle period of MDFG ;
P_x0 ← (0, 1) ; /* 1.1 find P_x0 */
P_y0 ← CCW vector of all delays; /* 1.2 find P_y0*/
f_x = { k | (0, k) is smallest (0, x) delays }; /* 1.3 find f_x*/
f_y = ⌈c / (T·f_x)⌉; /* 1.4 find f_y*/
P_x = f_x · P_x0; /* 1.5 find P_y and P_y*/
P_y = f_y · P_y0;
obtain basic partition with P_x, P_y;
/* Step 2. Call iterational retiming to transform the basic partition into a retimed partition; */
/* use r=(0,1) repeatedly to achieve full parallelism. */
Step 2.1 Apply r=(0,1) to any node that has all incoming edges with non-zero delays and at least one zero-delay outgoing edge.
Step 2.2 Since the resulting IFG is still realizable, if there are zero delay edges, go back to step 2.1.
```

The requirement for f_x is discussed in detail in section 3.1. We want f_x to be as large as possible. The larger f_x is, the smaller the prolog and epilog will be. Since $f_x \le min_k$, so we pick $f_x = min_k$. Once f_x is identified, we can find f_y with the given timing requirement T and the original cycle period c. Since we need to meet the timing constraint T,
$\Rightarrow T \ge \frac{c}{f_x \cdot f_y} \Rightarrow f_y \ge \frac{c}{T \cdot f_x} \Rightarrow f_y = \left\lceil \frac{c}{T \cdot f_x} \right\rceil$

THEOREM 3. *Let G_i be a realizable IFG, the iterational retiming algorithm transforms G_i to $G_{i,r}$, in at most $|V|$ iterations, such that $G_{i,r}$ is fully parallel.*

Proof: *After an iteration of the iterational retiming algorithm, the resulting IFG is still realizable. Successive iterations allow us to modify all zero delay edges to non-zero ones, obtaining a fully parallel IFG. After each iteration, all outgoing edges of at least one new node will not have any zero delay. After at most $|V|$ iterations, full-parallelism is achieved.*

To show how the algorithms work, we give an example. Figure 4 shows an example nested loop as well as the MDFG representation of the loop. Since $min_k = 2$, we will choose $f_x = 2$. Assume the timing requirement for iteration period is 1/2, we will choose $f_y = 2$. Figure 5(a) shows the iteration space with basic partition and the IFG representation of the basic partition. We can see from the IFG shown in Figure 5(a) that there are two zero delays, namely $c \rightarrow a$ and $d \rightarrow b$. Figure 5(b) shows the iteration space and the IFG after performing iteration retiming with $r(c) = (0, 1)$. Figure 5(c) shows the iteration space and the IFG after performing iteration retiming with $r(d) = (0, 1)$. From the iteration space in figure 5(c), we can see that all the iterations in each partition are independent and can be executed in parallel. Hence the retimed partition is reached. From the IFG in figure 5(c), we can see that there is no zero delays edges. Hence all the nodes(iterations) can be executed in parallel. With this character, iterational retiming is able to reduce iteration period and achieve higher level of parallelism.

```
for i=0 to n  do
   for j=0 to m do
       A[i,j] = B[i−1,j+1] + B[i,j−2];
       B[i,j] = A[i,j] + 5;
   end for
end for
```

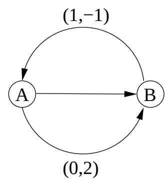

Figure 4: A nested loop and its MDFG.

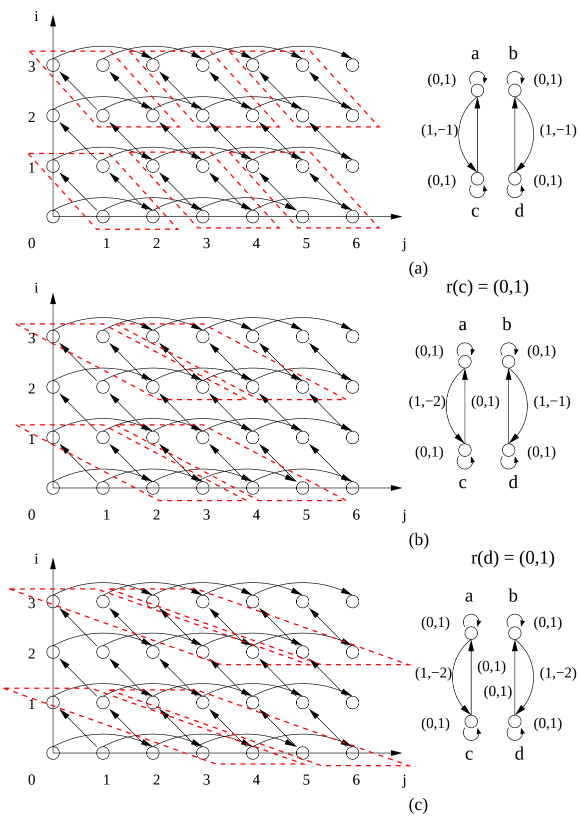

Figure 5: (a) Basic partition and IFG (b) First retimed partition and retimed IFG (c) Second retimed partition and retimed IFG

Algorithm 3.1 takes $O(|E| f_x f_y)$ to execute, where $|E|$ is the number of edges and f_x and f_y are the partition factors. For algorithm 3.2, in step 1, it takes $O(|V|)$ to find the cycle period, $O(|E|)$ to find P_{y0}, and $O(|E|)$ to find f_x. So it takes $O(|V| + |E|)$ to execute step 1. In step 2, it takes at most $|V|$ iterations, and each iteration takes at most $O(|E|)$ time. So it takes $O(|V||E|)$ to execute step 2. As a result, algorithm 3.2 takes $O(|V||E|)$ to complete.

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). 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, with timing constraint given as target iteration period, we compare the iteration periods of the initial loops, the iteration periods for the transformed loops obtained by loop unfolding, the iteration periods of loops obtained by basic partition, and the iteration periods of the transformed loops obtained by iterational retiming. The results are shown in table 2. In the table 2, columns "Init.", "Unfold.", "Parti." and "ITER-RE.", represent the iteration periods of the initial loops, the iteration periods of the unfolded loops, the iteration periods of the loops after basic partition, and the iteration periods of the loops after iterational retiming, respectively. Iteration periods in the table are average iteration periods. For unfolded loops, iteration periods are obtained by dividing the cycle periods of the unfolded loops by unfolding factor. For loops with partitions, iteration periods are obtained by dividing the cycle periods of the whole partition by the number of iterations inside a partition. For each target iteration period, the partition factors f_x and f_y are calculated first. Unfolding is done with unfolding factor equals to $f_x \times f_y$. From the results shown in table 2, we can see that for all the given timing constraints, iterational retiming can always achieve the target iteration periods. 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 iterational retiming with other techniques. For example, compared to loop unfolding, iterational retiming reduces iteration period by 88%.

For the results shown in table 3, we apply software pipelining to each input MDFG first. Column "Soft. Pipelin." represents the iteration period after applying software pipelining at instruction level. We can see that the initial iteration periods are all smaller or equal to the initial iteration period from table 2. The best software pipelining at instruction level can achieve is to reduce iteration period to 1 as shown in table Table 3. Again, column "ITER-RE" shows that iterational retiming can always achieve the given timing constraint consistently. From this table, we can see that after applying instruction-level software pipelining first, partition factor f_x and f_y is generally smaller then partition factors f_x and f_y shown in table table 2. In practice, we can apply instructional-level software pipelining first, and then apply iteration-level retiming, to give a smaller partition size as well as smaller prolog and epilog. The last row "Iter-re Avg Impv" also gives the average improvement obtained by comparing iterational retiming with other techniques. Compared to software pipelining, iterational retiming reduces iteration period by 87%.

Bench.	**Iteration Period (cycles)**					
Iteration Period = 1						
Bench.	f_x	f_y	Init.	Unfold.	Parti.	**ITER-RE.**
IIR	1	5	5	2.5	2.5	**1**
WDF	6	1	6	1	1	**1**
FLOYD	1	10	10	10	1	**1**
DPCM	1	5	5	2.5	2.5	**1**
2D(1)	1	9	9	2.8	2.8	**1**
2D(2)	1	4	4	4	1	**1**
MDFG1	1	7	7	7	7	**1**
MDFG2	1	10	10	10	7.3	**1**
Iteration Period = 1/2						
Bench.	f_x	f_y	Init.	Unfold.	Parti.	**ITER-RE.**
IIR	1	10	5	2.5	2.5	**0.5**
WDF	12	1	6	0.5	0.5	**0.5**
FLOYD	1	20	10	10	0.5	**0.5**
DPCM	1	10	5	2.5	2.5	**0.5**
2D(1)	1	18	9	2.2	2.2	**0.5**
2D(2)	1	8	4	4	0.5	**0.5**
MDFG1	1	14	7	7	7	**0.5**
MDFG2	1	20	10	10	7.1	**0.5**
Iteration Period = 1/4						
Bench.	f_x	f_y	Init.	Unfold.	Parti.	**ITER-RE.**
IIR	1	20	5	2.2	2.2	**0.25**
WDF	24	1	6	0.25	0.25	**0.25**
FLOYD	1	40	10	10	0.25	**0.25**
DPCM	1	20	5	2.2	2.2	**0.25**
2D(1)	1	36	9	2.2	2.2	**0.25**
2D(2)	1	16	4	4	0.25	**0.25**
MDFG1	1	28	7	7	7	**0.25**
MDFG2	1	40	10	10	7	**0.25**
Avg. Iter. Period			7	4.85	2.89	0.58
Iter-re Avg Impv			**92%**	**88%**	**80%**	

Table 2: Comparison of iteration period among list scheduling, loop unfolding, basic partition, and iterational retiming .

From our experiment results, we can clearly see iterational retiming technique can do much better in increasing parallelism and timing performance on nested loops than loop unfolding, software pipelining and basic partition.

5. CONCLUSION

In this paper, we propose a new loop transformation technique, iterational retiming. Iterational retiming can achieve any given timing constraint with minimum overhead. Theorems and algorithms are proposed to implement the iterational retiming technique. We believe iterational retiming is a promising technique and can be applied to different fields for nested loop optimization.

Bench.	**Iteration Period (cycles)**					
Iteration Period = 1						
Bench.	f_x	f_y	Soft. Pipelin.	Retime Unfold.	Retime Parti.	**ITER -RE.**
IIR	1	2	2	2	2	**1**
WDF	1	1	1	1	1	**1**
FLOYD	1	8	8	8	1	**1**
DPCM	1	2	2	2	2	**1**
2D(1)	1	1	1	1	1	**1**
2D(2)	1	4	4	4	1	**1**
MDFG1	1	7	7	7	7	**1**
MDFG2	1	10	10	10	7.3	**1**
Iteration Period = 1/2						
Bench.	f_x	f_y	Soft. Pipelin.	Retime Unfold.	Retime Parti.	**ITER -RE.**
IIR	1	4	2	2	2	**0.5**
WDF	2	1	1	0.5	0.5	**0.5**
FLOYD	1	16	8	8	0.5	**0.5**
DPCM	1	4	2	2	2	**0.5**
2D(1)	1	2	1	1	1	**0.5**
2D(2)	1	8	4	4	0.5	**0.5**
MDFG1	1	14	7	7	7	**0.5**
MDFG2	1	20	10	10	7.1	**0.5**
Iteration Period = 1/4						
Bench.	f_x	f_y	Soft. Piplelin.	Retime Unfold.	Retime Parti.	**ITER -RE.**
IIR	1	8	2	2	2	**0.25**
WDF	4	1	1	0.25	0.25	**0.25**
FLOYD	1	32	8	8	0.25	**0.25**
DPCM	1	8	2	2	2	**0.25**
2D(1)	1	4	1	1	1	**0.25**
2D(2)	1	16	4	4	0.25	**0.25**
MDFG1	1	28	7	7	7	**0.25**
MDFG2	1	40	10	10	7	**0.25**
Avg. Iter. Period			4.38	4.32	2.61	0.58
Iter-re Avg Impv			**87%**	**87%**	**78%**	

Table 3: Comparison of iteration period among software pipelining, retimed loop unfolding, retimed basic partition, and iterational retiming.

6. 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] L.-F. Chao, A. S. LaPaugh, and E. H.-M. Sha. Rotation scheduling: A loop pipelining algorithm. *IEEE Trans. on Computer-Aided Design*, 16(3):229–239, March 1997.

[4] L.-F. Chao and E.-M. Sha. Rate-optimal static scheduling for dsp data-flow programs. *IEEE Third Great lakes Symposium on VLSI*, pages 80–84, March 1993.

[5] L. Lamport. The parallel execution of do loops. *Communications of the ACM SIG-PLAN*, 17:82–93, FEB. 1991.

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

[7] N. Passos and E. Sha. Full parallelism of uniform nested loops by multi-dimensional retiming. *Internal conference on Parallel Processing*, 2:130–133, Aug. 1994.

[8] M. Renfors and Y. Neuvo. The maximum sampling rate of digital filters under hardware speed constraints. *IEEE Transactions on Cirtuits and Systems*, pages 196–202, March 1981.