Journal of Embedded Computing 3 (2007/2009) 29–37
DOI 10.3233/JEC-2009-0076
IOS Press

Optimizing parallelism for nested loops with iterational and instructional retiming[1]

Chun Jason Xue[a,*], Zili Shao[b], MeiLin Liu[c], Mei Kang Qiu[d] and Edwin H.-M. Sha[e]
[a]*City University of Hong, Kowloon, Hong Kong*
[b]*Hong Kong Polytechnic University, Kowloon, Hong Kong*
[c]*Wright State University, Dayton, OH, USA*
[d]*University of New Orleans, New Orleans, LA, USA*
[e]*University of Texas at Dallas, TX, USA*

Abstract. Embedded systems have strict timing and code size requirements. Retiming is one of the most important optimization techniques to improve the execution time of loops by increasing the parallelism among successive loop iterations. Traditionally, retiming has been applied at instruction level to reduce cycle period for single loops. While multi-dimensional (MD) retiming can explore the outer loop parallelism, it introduces large overheads in loop index generation and code size due to loop transformation. In this paper, we propose a novel approach, that combines iterational retiming with instructional retiming to satisfy any given timing constraint by achieving full parallelism for iterations in a partition with minimal code size. The experimental results show that combining iterational retiming and instructional retiming, we can achieve 37% code size reduction comparing to applying iteration retiming alone.

1. Introduction

Constant advance in gate capacity and reduction in chip feature size made it possible to implement system-on-chip (SoC) that allows parallel embedded systems. Parallel embedded systems, typically with VLIW or multi-core architecture, are becoming more widely used in 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. There are usually stringent requirements in timing performance and code size for these applications. How to design parallel embedded systems that conforms timing performance requirements while minimizing code size is an interesting research topic. This paper combines iterational retiming with instructional retiming, to achieve given timing requirements for nested loops with minimal code size overhead.

There has been a lot of research on program transformations to enhance parallelism for loops. Many forms of program transformations have been proposed, a catalog of which can be found in a survey by Bacon, Graham, and Sharp [3]. Numerous techniques have been proposed for one-dimensional loops [1,4,5,8]. Renfors and Neuvo have proved that there is a lower bound of iteration period for one-dimensional loops [10]. Optimal scheduling for one-dimensional loops could reach this lower bound [5], 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 given 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 [2,7], which achieves higher level of parallelism for nested loops by changing the execution sequence of the nested loops. This sequence of execution is commonly associated

[1]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.
*Corresponding author. E-mail: jasonxue@cityu.edu.hk

ISSN 1740-4460/07-09/$17.00

```
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
```

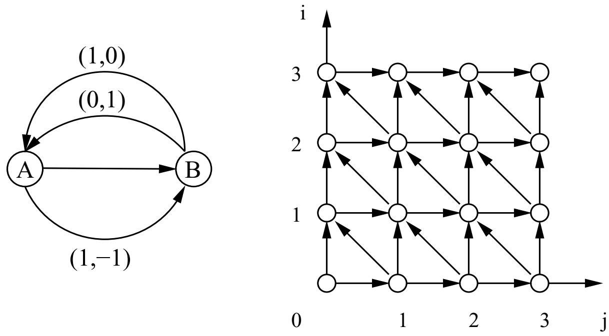

Fig. 1. A nested loop, its MDFG and its iteration space.

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 boundaries 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.

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 minimal overhead. Two main techniques are applied in iterational retiming, namely loop partitioning and retiming. Various techniques have been proposed for loop partitioning [6, 11]. most of the loop partitioning algorithm are targeting better data locality. In our algorithm, partition is used to increase parallelism. Retiming [8,9] 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 iterational retiming can always satisfy the timing requirement, it is best to be combined with instructional retiming. We show that when instructional retiming is combined with iterational retiming, code size is reduced 37% in average, while achieving the same given timing performance constraint.

The remainder of this paper is organized as follows. Section 2 introduces basic concepts and definitions. The algorithm for instructional timing is presented in Section 3. The algorithm for iterational retiming is proposed in Section 4. Section 5 proposes the algorithm that combines iterational retiming and instructional retiming. Experimental results and concluding remarks are provided in Sections 6 and 7, 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 Fig. 1.

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 Fig. 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

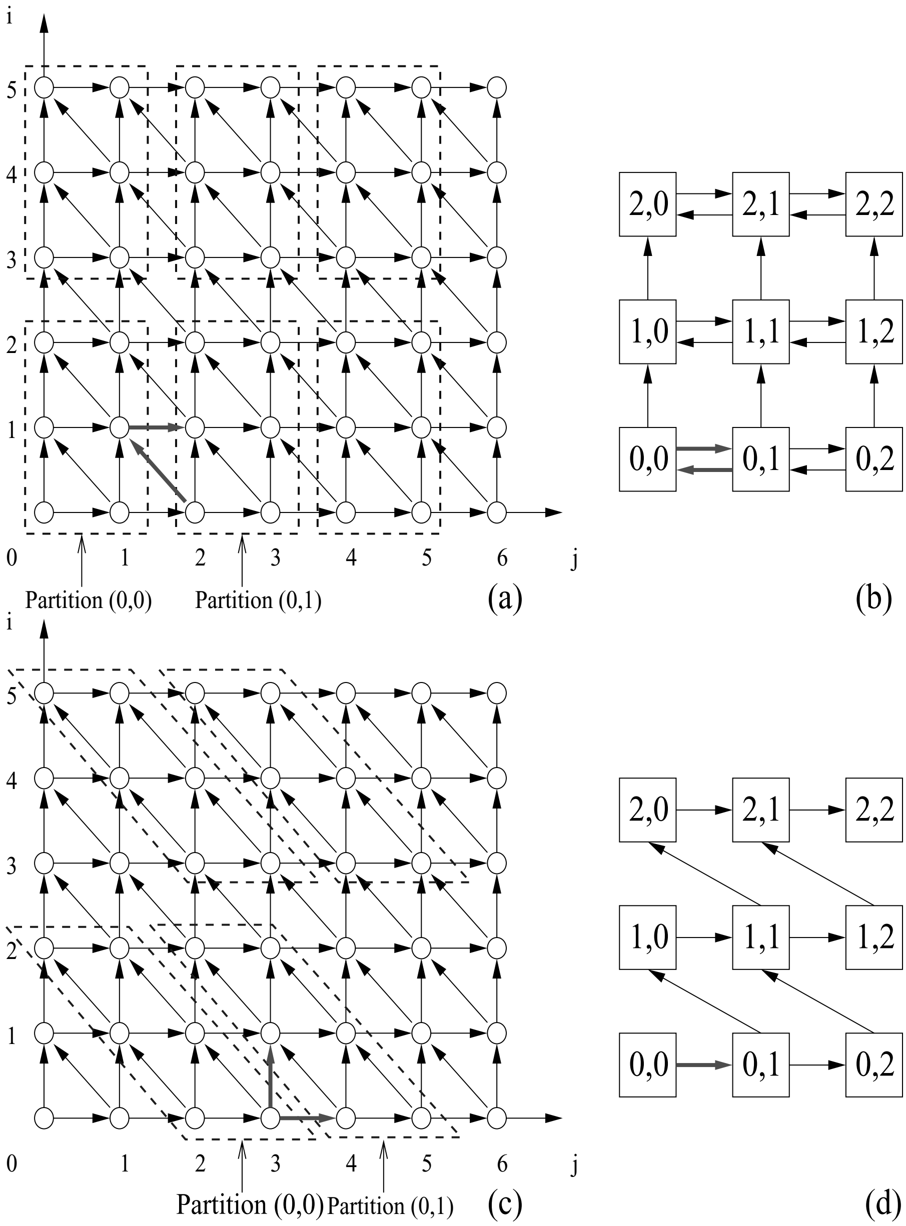

Fig. 2. (a) An illegal partition of the iteration space (b) Partition dependency graph (c) A legal partition (d) Partition dependency graph.

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 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 [8] 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 $e(u \rightarrow v) \in E_r$ 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 need to be carefully chosen to ensure there is no two-way dependency so that the partition is a legal partition. For example, consider the iteration space in Fig. 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 Fig. 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 Fig. 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.

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 Fig. 2(c) is shown in Fig. 3. Edges with (0,0) delay are shown in thicker line.

We use d(e) = (d.x, d.y) as a general formulation of any delay shown in a two-dimensional DFG (2DFG).

An equivalent *Partition Dependence Graph* (*PDG*) of an IFG G_i is the directed acyclic graph shown the dependencies between copies of nodes representing the partitions. Examples of PDGs are shown in Fig. 2, where (b) is the partition dependence graph for partition shown in (a), and (d) is the partition dependence graph for (c). Legal partition requires that there is no cycle in the partition dependence graph, otherwise, we will not able to schedule such a partition.

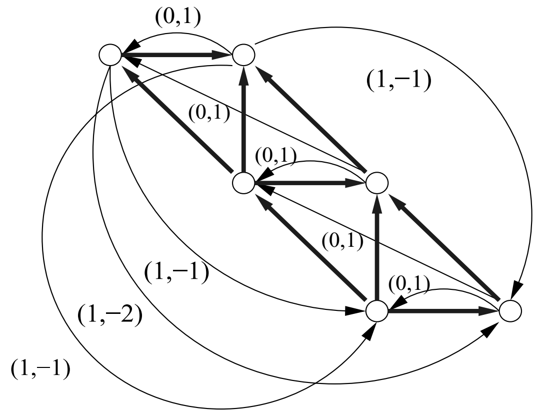

Fig. 3. IFG for the partition in Fig.2 (c).

3. The instructional retiming for nested loops

We propose an algorithm to do maximum instructional retiming while maintaining the original execution sequence in this section.

The instructional retiming we are applying in this setting is unique. We are retiming along one dimension of a multi-dimensional loop. This is because we wish to maintain either row-wise or column-wise execution after the retiming. The problem is equivalent to removing all the (i,j)-delays, where both i and j are none zero values, and producing a retiming such that all the edges are non-zero-delay edges if possible. If there is at least one (i,j)-delay, we can always find a retiming solution because the cycle becomes a DAG after removing (i,j)-delay edge. If there is no (i,j)-delay, then no retiming solution exists because the total delay of a cycle is a constant according to retiming property. In this case, the total delay of the cycle is always (0,k). According to retiming theory, DAG can always be retimed such that any edge has at least one delay. In the other words, a cycle with any (i,j) delay, $i > 0$, can be fully parallelized using schedule vector $\mathbf{s} = (1,0)$. Thus, we can directly apply one-dimensional retiming to minimize the cycle period after removing the (i,j)-delay edges. Detail of our instructional retiming algorithm is shown in 3.1.

4. 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.

Algorithm 3.1 INSTRUCTIONAL-RETIME Algorithm

Require: MDFG $G = \langle V, E, \mathbf{d}, t\rangle$, schedule vector‘ $\mathbf{s}$.
Ensure: Retiming function $\mathbf{r}(v)$, retimed MDFG $G_r = \langle V, E, \mathbf{d}_r, t\rangle$, minimum cycle period $\mathbf{c}$.
/* Check the legality of $\mathbf{s}$. */
if d$(e)\cdot$ $\mathbf{s} < 0$
Report: $\mathbf{s}$ is unfeasible.
Break.
end if
/* For the feasible clock-period test algorithm */
Compute $D(u, v)$ and $T(u, v)$ for any two nodes u and v on G'.
Sort the elements in the range of $T(u, v)$.
for all Elements in the range of $T(u, v)$ **do**
Binary search to find the minimum achievable cycle period c.
/* Determine if cycle period c is feasible */
$r \leftarrow (0, 1)$; $s \leftarrow (1, 0)$;
/* Remove the edges that are not parallel to $\mathbf{r}$. */
$E' \longleftarrow E - \{e \mid \mathbf{d}(e) \not\parallel \mathbf{r}, \forall e \in E\}$
$\mathbf{d}'(e) \longleftarrow \mathbf{d}(e)/\mathbf{r}, \forall e \in E'$
/* The feasible clock-period test algorithm */
Compute $D(u, v)$ and $T(u, v)$ for any two nodes u and v on G'.
Construct the constraint graph according to the legality and feasible conditions of retiming with the desired cycle period c.
Use the shortest path algorithm to find a solution.
if There exists a retiming r' of G' such that $\Phi(G'_{r'}) \leqslant c$ **then**
/* Update the retimed graph */
$\mathbf{r}(v) \longleftarrow r'(v) \cdot \mathbf{r}(v), \forall v \in V$
c feasible; Use retiming $\mathbf{r}$ to retime the MDFG G.
else
Error: Unfeasible cycle period.
end if
end for

4.1. 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 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 $d(c) \geqslant (1, -\infty)$. The cycles with delays $d(c) \geqslant (1, -\infty)$ can be easily retimed to be fully parallel by using $r = (0, 1)$. But for cycles

Algorithm 4.1 ITERATIONAL-RETIME

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; */
$c \leftarrow$ cycle period of MDFG ;
$P_{x0} \leftarrow (0, 1)$; /* 1.1 find P_{x0} */
$P_{y0} \leftarrow$ 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*/
$f_y = \left\lceil \frac{c}{T \cdot f_x} \right\rceil$; /* 1.4 find f_y*/
$P_x = f_x \cdot P_{x0}$; /* 1.5 find P_y and P_y*/
$P_y = f_y \cdot P_{y0}$;
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.

with delays $d(c) = (0, y)$, y must be $\geqslant$ 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 $d(c) \geqslant n(c)$, it means that $d(c) = (0, y)$, $y \geqslant n(c)$.

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)), $0 \leqslant i < f_x$, 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 $\leqslant (0, 1)$. 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 $\leqslant (0, 1)$ 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 $f_x \leqslant min_k$, for each cycle c in IFG, d(c) must be $\geqslant n(c)$.*

Proof. Considering a delay (0, b) in DFG, whose b must be $\geqslant f_x$ from the condition of the theorem.
Case 1: b = (0, z $\times$ min_k), $z \geqslant 1$. This will cause self cycles in IFG with (0, z) delay, $z \geqslant 1$. So $d(c) \geqslant n(c)$.
Case 2: $b > f_x$. Based on the property described previously, for a node i in IFG, i $\rightarrow$ (i + b) mod f_x has a delay (i + b) div f_x. It is obvious that (i + b) $> f_x$, so the delay $\geqslant (0, 1)$. Thus, $d(c) \geqslant n(c)$ for each c. (actually this case 2 can cover case 1. when you just use $b \geqslant f_x$).

As a result of the above theorems, we know that let $f_x \leqslant min_k$, 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.

4.2. The iterational retiming technique

In this section, the iterational retiming algorithm is presented and explained. The complexity of the algorithm is given at the end of this section.

The requirement for f_x is discussed in detail in Section 4.1. We want f_x to be as large as possible. The larger f_x is, the smaller the prolog and epilog will be. Since $f_x \leqslant min_k$, 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,

$$\Rightarrow T \geqslant \frac{c}{f_x \cdot f_y}$$

$$\Rightarrow f_y \geqslant \frac{c}{T \cdot f_x}$$

$$\Rightarrow f_y = \left\lceil \frac{c}{T \cdot f_x} \right\rceil$$

Algorithm 5.1 COMBINE-RETIME
Require: MDFG $G = \langle V, E, d, t\rangle$, timing requirement T
Ensure: A retimed MDFG that meets timing requirement with smallest overheads
/* Step 1. apply instructional retiming */
if apply INSTRUCTIONAL-RETIME algorithm with $s = (0, 1)$ can satisfy T **then**
return $s = (0, 1)$;
else if apply INSTRUCTIONAL-RETIME algorithm with $s = (1, 0)$ can satisfy T **then**
return $s = (1, 0)$;
else
if $s = (1, 0)$ is legal and INSTRUCTIONAL-RETIME with $s = (1, 0)$ give a smaller cycle period **then**
apply loop index interchange;
end if
/* Step 2. apply iterational retiming reach timing requirement T */
call ITERATIONAL-RETIME algorithm;
end if

```
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
```

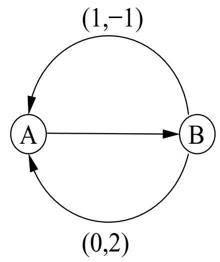

Fig. 4. A nested loop and its MDFG.

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 the nested loop as well as the MDFG representation of the loop for the example. The original cycle period c of this MDFG is 2, which is given by the longest path without delay from node A to node B. Since $min_k = 2$, we will choose $f_j = 2$. Assuming a timing requirement for iteration period T is 1/2, we have

$$f_i = \left\lceil \frac{c}{T \cdot f_j} \right\rceil = \left\lceil \frac{2}{\frac{1}{2} \cdot 2} \right\rceil = 2.$$

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 Fig. 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 Fig. 5(c), we can see that all the iterations in each partition are independent and can be executed in parallel. Hence a retimed partition is reached. From the IFG in Fig. 5(c), we can see that there is no zero delay edges. All the nodes, i.e., iterations, can be executed in parallel. With this character, iterational retiming is able to reduce iteration period and achieve higher level of parallelism.

For algorithm 4.1, 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 4.1 takes $O(|V||E|)$ to complete.

5. Combining iterational and instructional retiming

The COMBINE-RETIME algorithm is presented in this section. It combines instructional retiming with iterational retiming to optimize nested loops for both timing performance and code size.

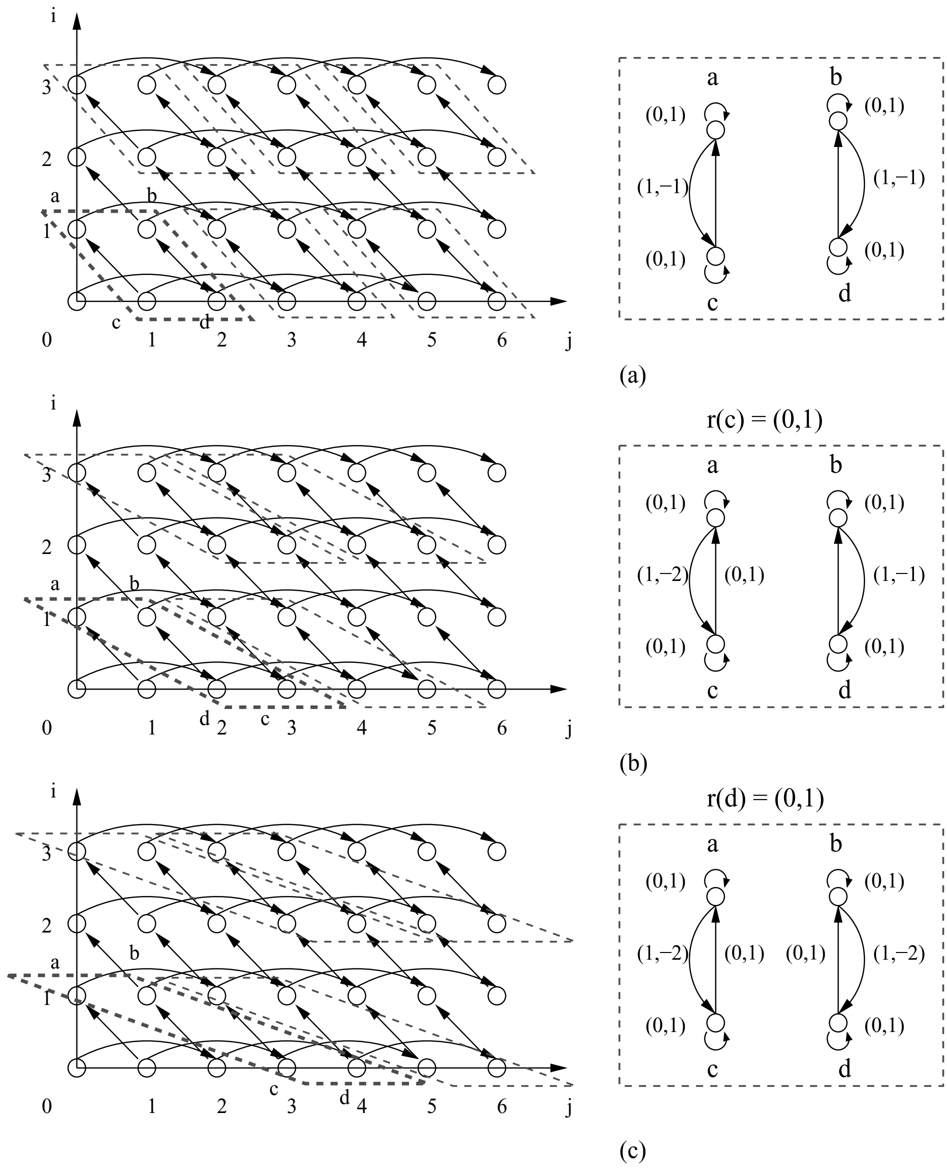

Fig. 5. (a) Basic partition and IFG. (b) First retimed partition and retimed IFG. (c) Second retimed partition and retimed IFG.

The COMBINE-RETIME algorithm first applies the INSTRUCTIONAL-RETIME algorithm in one dimension to minimize the cycle period. Row-wise execution sequence, i.e. schedule vector $s = (0{,}1)$ is first attempted. Then column-wise execution sequence is attempted if schedule vector $s = (1{,}0)$ is legal. When both instructional retiming failed to meet the required timing constraint T, ITERATIONAL-RETIME algorithm is called to perform iterational retiming. When column-wise execution sequence is legal and it gives a smaller cycle period after iterational retiming, loop index interchange is applied before performing iteration retiming. Applying instructional retiming reduces iteration period before iterational retiming is applied. Smaller partition size is needed in iterational retiming process, hence code size is reduced given the same timing performance requirement.

6. 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

Table 1
Comparison of code size between iterational retiming only and combined retiming.

Bench.	Code Size (Iterations)					
	f_x	f_y	ITER-RE.	f_x	f_y	COM-RE.
Iteration Period = 1						
IIR	1	5	5	1	2	2
WDF	6	1	6	1	1	1
FLOYD	1	10	10	1	8	8
DPCM	1	5	5	1	2	2
2D(1)	1	9	9	1	1	1
2D(2)	1	4	4	1	4	4
MDFG1	1	7	7	1	7	7
MDFG2	1	10	10	1	10	10
Iteration Period = 1/2						
IIR	1	10	10	1	4	4
WDF	12	1	12	2	1	2
FLOYD	1	20	20	1	16	16
DPCM	1	10	10	1	4	4
2D(1)	1	18	18	1	2	2
2D(2)	1	8	8	1	8	8
MDFG1	1	14	14	1	14	14
MDFG2	1	20	20	1	20	20
Iteration Period = 1/4						
IIR	1	20	20	1	8	8
WDF	24	1	24	4	1	4
FLOYD	1	40	40	1	32	32
DPCM	1	20	20	1	8	8
2D(1)	1	36	36	1	4	4
2D(2)	1	16	16	1	16	16
MDFG1	1	28	28	1	28	28
MDFG2	1	40	40	1	40	40
Avg. Iter.			16.3			10.2
Avg Impv			37%			

Response Filter), 2D (the Two Dimensional filter), Floyd (Floyd-Steinberg algorithm), and DPCM (Differential Pulse-Code Modulation device).

Table 1 shows the code size comparison between applying iterational retiming alone and applying both iterational retiming and instructional retiming transformation, while achieving the same timing performance. In this table, code size is measure in the number of iterations included in each partition, which equals $f_x \times f_y$. Column "ITER-RE" represents the code size by applying iterational retiming alone. Column "COM-RE" represents the code size by applying both instructional retiming and iterational retiming. From this table, we can see that by applying instructional retiming before iterational retiming, we can reduce code size in an average of 37%.

From our experiment results, we can clearly see that combining iterational retiming with instructional retiming can reduce code size significantly while achieving the same timing performance constraints.

7. Conclusion

In this paper, we propose a new loop transformation approach that combines iterational retiming with instructional retiming to optimize both timing performance and code size. We believe iterational retiming is a promising technique and can be applied to different fields for nested loop optimization. Combining instructional retiming and iterational retiming, we can achieve timing performance requirement for nested loops with minimal code size.

References

[1] A. Aiken and A. Nicolau, Optimal loop parallelization, *ACM Conference on Programming Language Design and Implementation* (1988), 308–317.

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

[3] D. Bacon, S. Graham and O. Sharp, Compiler transformations for high-performance computing, *Computing Srveys* (1994), 345–420.

[4] 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) (March 1997), 229–239.

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

[6] F. Chen and E.-M. Sha, Loop scheduling and partitions for hiding memory latencies, *International Symposium on System Synthesis*, 1999.

[7] L. Lamport, The parallel execution of do loops, *Communications of the ACM SIG-PLAN* **17** (February) (1991), 82–93.

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

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

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

[11] Z. Wang, Q. Zhuge and E.-M. Sha, Scheduling and partitioning for multiple loop nests, *International Symposium on System Synthesis* (October 2001), 183–188.