Rotation Scheduling: A Loop Pipelining Algorithm

Liang-Fang Chao, *Member, IEEE*, Andrea S. LaPaugh, *Member, IEEE*, and Edwin Hsing-Mean Sha, *Member, IEEE*

***Abstract*— We consider the resource-constrained scheduling of loops with interiteration dependencies. A loop is modeled as a *data flow graph (DFG)*, where edges are labeled with the number of iterations between dependencies. We design a novel and flexible technique, called *rotation scheduling*, for scheduling cyclic DFG's using loop pipelining. The rotation technique repeatedly transforms a schedule to a more compact schedule. We provide a theoretical basis for the operations based on retiming. We propose two heuristics to perform rotation scheduling and give experimental results showing that they have very good performance.**

***Index Terms*— High-level synthesis, loop pipelining, parallel compiler, retiming, scheduling.**

I. Introduction

For real-time or high-performance computing, a synthesis system needs to have the ability to optimize the execution rate of a design. Since loops are usually the most time-critical parts of an application, the parallelism embedded in the repetitive pattern of a loop needs to be explored. This paper proposes a generic technique for the scheduling of loops when resource constraints are present.

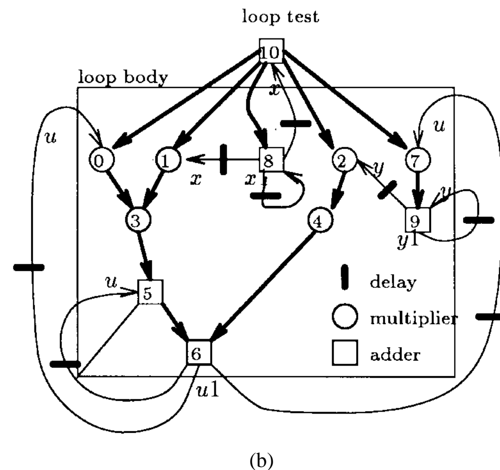
A loop can be modeled as a *data flow graph (DFG)*, as shown in Fig. 1. Each computation in the loop is represented as a node and a precedence relation as an edge. Each edge has a number of *delays* (registers). This data-flow graph model is widely used in many fields, for example, in circuitry [13], in digital signal processing [9], and program descriptions [1], [10]. We consider not only DFG's with interiteration dependencies but also *cyclic* DFG's, where precedence constraints might form cycles.

Scheduling cyclic DFG's with resource constraints are more difficult than scheduling *acyclic* DFG's with resource constraints. The loops are usually pipelined in order to increase the execution rate, where the execution periods of several iterations are overlapped. Loop winding [7] was proposed to pipeline loops with acyclic DFG's, where theoretically the performance can be made arbitrarily good. However, when pipelining cyclic DFG's, the cycles confine the depth and freedom of loop pipelining. In contrast to acyclic DFG's, cycles in a DFG provide bounds on the improvement we can

To solve $y'' + 3xy' + 3y = 0$.

```
while (x < a) do
    x1 = x + dx;
    u1 = u - (3 * x * u * dx) - (3 * y * dx);
    y1 = y + u * dx;
    x = x1;   u = u1;   y = y1;
end
```

(a)



(b)

Fig. 1. Differential equation solver.

achieve by pipelining. In this paper, we propose a generic technique to optimize a cyclic DFG under resource constraints.

A *static* schedule of a loop is repeatedly executed for the loop. Edges without delays represent intra-iteration precedence relations, for example the thick arcs in Fig. 1(b). Thus, a static schedule must obey the precedence relation defined by the *directed acyclic graph (DAG)* consisting of edges without delays in a DFG. The path with the longest total computation time in the DAG is the *critical path*, the length of which defines the *iteration period* of the DFG.

The technique of *retiming* [13] is an effective technique to optimize a DFG in order to obtain an equivalent DFG with shorter iteration period by rearranging delays. Retiming provides a simple model for loop pipelining and efficient algorithms. Loop pipelining wraps a loop around, overlapping operations from different iterations to create a more compact schedule. It basically reorganizes a loop and changes some initial values of a loop.

Previous work on loop pipelining for loops with cyclic dependencies appears in several high-level synthesis systems

Manuscript received June 22, 1993; revised December 22, 1996. This paper was recommended by Associate Editor, M. McFarland. This work was supported in part by DARPA/ONR Contract N00014-88-K-0459 and by the National Science Foundation under Grant MIP90-23542.

L.-F. Chao is with the Department of Electrical and Computer Engineering, Iowa State University, Ames, IA 46556 USA.

A. S. LaPaugh is with the Department of Computer Science, Princeton University, Princeton, NJ 08544 USA.

E. H.-M. Sha is with the Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556 USA.

Publisher Item Identifier S 0278-0070(97)04740-4.

0278–0070/97$10.00 © 1997 IEEE

CS	Mult	Adder
1	–	10
2	1	8
3	0	–
4	3	–
5	2	5
6	4	–
7	7	6
8	–	9

(a)

CS	Mult	Adder
2	1	8
3	0	**10**
4	3	–
5	2	5
6	4	–
7	7	6
8	–	9

(b)

CS	Mult	Adder
3	0	10
4	3	**8**
5	2	5
6	4	–
7	7	6
8	**1**	9

(c)

Fig. 2. Two down-rotations of size one for unit-time operations.

[8], [12], [17], [18], [23] and parallel compilers for very-long-instruction-word (VLIW) machines [6], [10]. Detailed comparisons of our approach with these methods are discussed in Section VII. Work on high-level synthesis mostly focuses on the innermost loops without conditional statements and is oriented for digital signal processing (DSP) applications. The algorithms by Lee *et al.* [12] and in MARS [23] are designed for time-constrained scheduling. Schedules are first generated to satisfy time constraints, and operations are rescheduled selectively to minimize the amount of resources. We consider resource-constrained scheduling to maximize the execution rate. Percolation-based scheduling [15], [17] uses a set of transformations to merge operations into control steps, and the pipelined loop body is obtained with incremental unfolding. Cathedral II [8] is especially designed for DSP applications, where resource constraints are not considered during the retiming phase.

Our loop pipelining algorithm improves a legal schedule and performs implicit retiming incrementally by the *rotation* technique. An existent schedule is partially rescheduled by rotation to obtain a shorter and valid schedule under resource constraints. The result of rotation retimes the DFG implicitly to naturally produce a pipeline schedule. The state of a sequence of rotations is recorded by a simple retiming (node-labeling) function. In fact, rotation is a generic technique that can be used to design a class of heuristic algorithms for loop pipelining. The two simple heuristics proposed give very good experimental results.

We use the differential equation solver in [16] as an example throughout the paper. The behavioral description and DFG are shown in Fig. 1. Here, we assume that additions and multiplications take one time unit, and a *control step (CS)*, also called *clock cycle*, consists of one time unit. For a resource of one multiplier and one adder, an optimal schedule for the DAG part of the DFG is shown in Fig. 2(a). Fig. 2(b) shows a more compact schedule, in which node 10 has been rotated down and then pushed up to its new position. This new schedule is a schedule of the retimed graph G_r, as shown in Fig. 3(a). Intuitively, node 10, which was a root in the original DFG, is *rotated down* into a leaf. The nodes 1 and 8 in Fig. 2(b) are rotated down and then rescheduled to their new positions in Fig. 2(c), which is an optimal schedule for this example. The retimed DFG is shown in Fig. 3(b). We provide theoretical foundations to support such movement of nodes, and methods to check the retimed DFG for rescheduling.

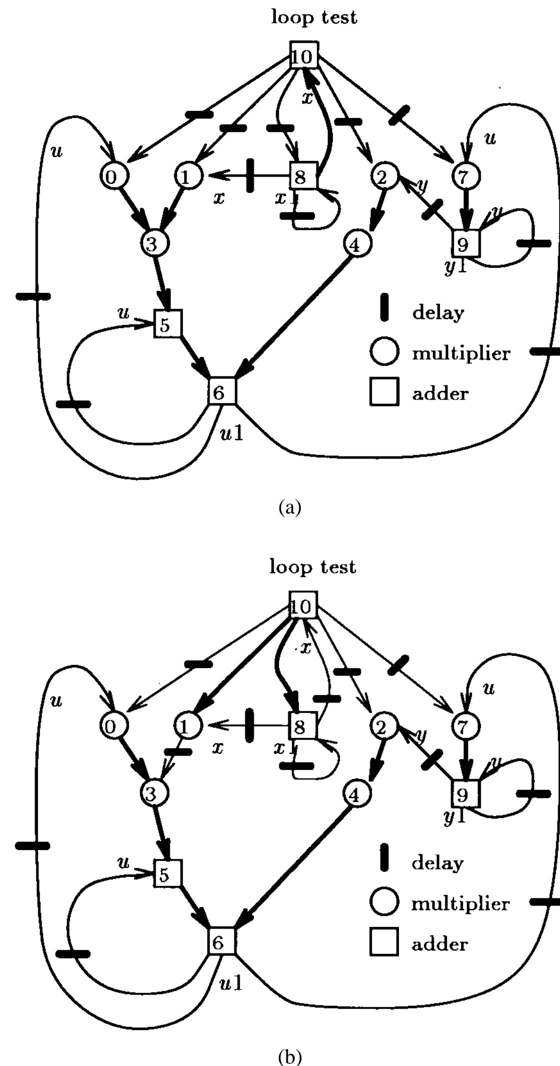


Fig. 3. Corresponding retimed graphs after rotations.

Rotation uses a DAG scheduling algorithm, like *list scheduling*, as a subroutine, and can be easily incorporated into an existing DAG scheduler. This method can handle chained op-

erations, multicycle operations, and pipelined functional units. Since only a part of the DFG is rescheduled in each rotation, computation time can be saved by only rescheduling the rotated part. Therefore, the system can perform more rotations, consider more retimed graphs, and find better solutions faster. The model of rotations by retiming simplifies the checking of precedence constraints without reconstructing a DFG after each rotation. There is no need to construct the retimed graphs in our procedure. The retimed graphs are drawn in this paper to help the presentation.

The next section defines *down* and *up* rotations. A basic rotation algorithm is presented in Section III on DFG's with single-cycle and chained operations. An efficient algorithm is presented to find a pipeline with a shallow depth from a schedule. In Section IV, the basic algorithm is then refined to handle multicycle operations and pipelined functional units. The concept of a *wrapped* schedule is also introduced there. A couple of heuristics are proposed in Section V. These heuristics give very good experimental results, which are presented in Section VI. Comparisons of our approach with many loop pipelining algorithms are discussed in Section VII. We believe that the rotation technique lays a good foundation for the resource-constrained scheduling of loops.

II. DEFINITIONS

A *data-flow graph (DFG)* is a directed weighted graph $G = (V, E, d, t)$ where V is the set of computation nodes, E is the edge set which defines the precedence relations from nodes in V to nodes in V, $d(e)$ is the number of delays for an edge e, and $t(v)$ is the computation time of a node v.

We define one *iteration* to be the execution of each node in V exactly once. An edge e from u to v with $d(e)$ delays means that the computation of node v at iteration j depends on the computation of node u at iteration $j - d(e)$. A *static* schedule of a loop is repeatedly executed for the loop. An edge without delays represents an intra-iteration precedence relation. A static schedule must obey the precedence relations defined by the subgraph consisting of edges without delays in a DFG. Thus, this subgraph must be a DAG. The path with the maximum total computation time of the DAG defines the *iteration period* of the DFG. The iteration period of the DFG is the length of the static schedule for the DAG without resource constraints.

The technique of retiming moves around delays in the following way. A delay is drawn from *each* of the incoming edges of v, and, then, a delay is pushed to *each* of the outgoing edges of v and vice versa. A retiming r of a DFG G is a function from V to integers [13]. The value $r(v)$ is the number of delays pushed through node v from the incoming edges to the outgoing edges.[1] Let $G_r = (V, E, d_r, t)$ be the DFG retimed by a retiming r from G, where $d_r(e) = d(e) + r(u) - r(v)$ for every edge e. A retiming r is *legal* if the value $d_r(e)$ is nonnegative for every edge e in E. Without loss of generality, we consider only *normalized* retiming functions.

[1] Note that our definition of retiming functions is slightly different from that by Leiserson and Saxe [13]. In our definition, $r(v)$ is positive if delays are pushed along the direction of edges. We think this definition is more natural, especially in loop scheduling.

A retiming function r is *normalized* if $\min_v r(v) = 0$. For any retiming function r', we can normalize it into r where $r(v) = r'(v) - \min_v r(v)$ for every v in V.

A set X of nodes in V is represented by a zero to one valued function X from V to $\{0, 1\}$. A node v belongs to set X if and only if $X(v) = 1$. This set representation is also used as a retiming function in our rotation operation.

The operation of *down-rotation* is defined as follows.

Definition 1: The down rotation of G on X pushes one delay from each of the incoming edges of X to each of the outgoing edges of X. The DFG G is transformed into DFG G_X after set X is rotated down.

When a node is rotated down, a delay is pushed from all of its incoming edges to all of its outgoing edges. Consider a simple down-rotation of node 10 in Fig. 1(b). This node, a root of the original DAG, becomes a leaf node in the new DAG, shown in Fig. 3(a). Thus, intuitively it is *rotated down*. The operation of *up rotation* on set X transforms G into G_{-X} by pushing one delay in the reverse direction. We say that a set X is *down rotatable* if X is a legal retiming for G. Not any subset in V is rotatable. It is not hard to show the following property.

Property 1: A set X is down rotatable if and only if every path from $V - X$ to X contains at least one delay.

For example, in Fig. 1(b), the sets $\{10, 8, 1\}$ and $\{10\}$ are down-rotatable sets, but $\{8, 1\}$, $\{1\}$, and $\{8\}$ are not. Fig. 3(a) shows the retimed DFG after $\{10, 8, 1\}$ is rotated down. The *up-rotatable* set is similarly defined. In this paper, we will focus on the properties of down rotations. Similar properties and algorithms can be derived for up rotations.

We define the composite of two retimings as $r_1 \circ r_2(v) = r_1(v) + r_2(v)$. The composite of a sequence of down rotations is the composite of the retimings of the down-rotation sets. Therefore, the composite of a sequence of rotations can be represented by a single retiming function.

The advantage of associating retiming functions with rotations is that the precedence constraints, captured by d_r, of a retimed graph can be easily examined from the original DFG and the retiming r. In some pipeline scheduling algorithms [8], [10], at each run, a precedence constraint graph has to be constructed or weights on graph edges have to be updated. Using a retiming function to model a sequence of rotations, no graphs or weights on graph edges are modified in order to capture precedence relations. Therefore, a lot of computation time can be spared.

The loop, represented by a DFG, is executed *in pipeline* if the execution periods of several iterations are overlapped. A static schedule describes a loop pipeline at its stable state. The length of a static schedule corresponds to the minimum cycle period of the loop pipeline, also called the *minimum initiation interval*.

If the nodes in a static schedule are from p different iterations, there are p pipeline stages. We call such p as the *depth* of a loop pipeline. Consider the retimed DFG in Fig. 3(b). There are two stages in the pipeline: the set of nodes with $r(v) = 1$ is in the first stage, i.e., $\{10, 8, 1\}$; the set of nodes with $r(v) = 0$ is in the second stage, i.e., the rest of nodes. We have the following property.

Property 2: Let r be a retiming function. The depth of a loop pipeline represented by a retiming function r is

$$1 + \max_v r(v) - \min_v r(v).$$

The depth of a normalized retiming function is $1 + \max_v r(v)$.

An algorithm is presented in the next section to reduce the depth of a given loop pipeline.

III. The Basic Algorithms

In this section, we design a technique to compact a given schedule of a DFG under resource constraints by the technique of rotations. The basic rotation algorithm works for control steps with chained operations. We refine the algorithm for more general models (multicycle operations and pipelined functional units) in the next section. After a sequence of rotations is performed, the depth of a loop pipeline might be too long. An efficient algorithm is presented in Section III-B to reduce the depth of a pipelined schedule.

Let $[l, r]$ be the set of integers $\{i \mid l \leq i \leq r\}$. A schedule s is a mapping from V to control steps such that the resource constraints are satisfied. The computation node v starts its execution at control step $s(v)$. The *DAG schedule* of a DFG G is *legal* if for every edge $(u, v) \in E$, $s(u) + t(u) \leq s(v)$ if $d(u, v) = 0$. A *static schedule* s is *legal* if there exists a retiming r such that schedule s is a legal DAG schedule for the retimed DFG G_r. The following lemma gives a characterization of *legal static schedule*.

Lemma 1: Let G be a DFG and s be a schedule of the DFG. s is a legal static schedule of G if and only if there exists a legal retiming r such that $s(u) + t(u) \leq s(v)$ if $d_r(u, v) = 0$.

We say that the retiming r satisfying this lemma *realizes* the schedule s. A schedule may be realized by several retimings. In Section III-B, we present an algorithm to find a retiming with a short pipeline depth for a given schedule.

A. The Basic Rotation Algorithm

For any legal DAG schedule of a DFG, we use the technique of rotation to compact a schedule to obtain a legal static schedule. For example, consider the DFG in Fig. 1(b). By list scheduling which uses the number of descendants as the weight of a node in the list, the DFG has a schedule of length eight [see Fig. 2(a)], which is an optimal DAG schedule. We can rotate down the nodes $X_1 = \{10\}$ in the first control step. Then, we try to push the node 10 up to its earliest possible control steps according to the precedence constraints of the DAG of G_{R_1}, where $R_1 = X_1$, as shown in Fig. 3(a). This is actually achieved by rescheduling the set of nodes rotated down, $\{10\}$, by list scheduling. We obtain a schedule of length seven as shown in Fig. 2(b). After another rotation of one control step, we reschedule the nodes $X_2 = \{1, 8\}$ according to the DAG of G_{R_2}, where $R_2 = R_1 \circ X_2$, as shown in Fig. 3(b). The optimal schedule in Fig. 2(c) is then obtained after two rotations of one control step. The retimings obtained from a sequence of rotations, e.g., R_2, are called *rotation functions*.

Fig. 4 shows the effect of rotation in a global view after the above two rotations. Fig. 4(b) shows the situation where rotations are performed without rescheduling. The prologue and epilogue are introduced, and the static schedule is a schedule for the DAG part of the retimed DFG G_r with $r(1) = r(8) = r(10) = 1$, as shown in Fig. 3(b). Fig. 4(c) shows the global view of the schedule in Fig. 2(c), where nodes 1, 8, and 10 are rescheduled according to r. Intuitively, when a node is rotated, each copy of the node is pushed up by one iteration to a location in the previous location, and the first copy of the node is pushed into the prologue.

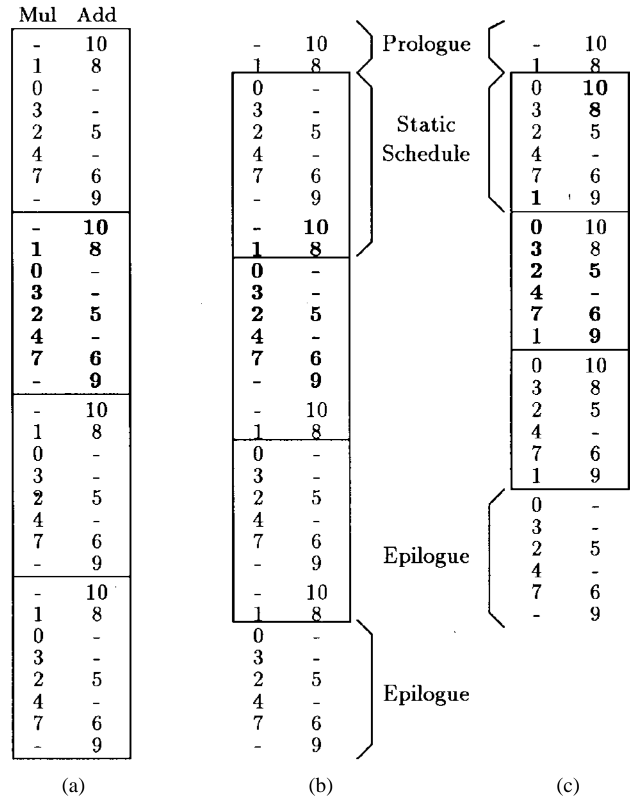
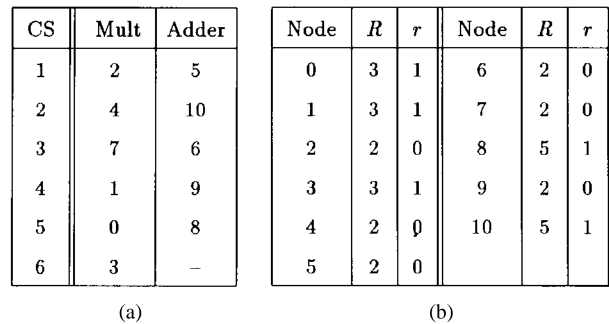


Fig. 4. Entire loop schedules after rotations.



CS	Mult	Adder
1	2	5
2	4	10
3	7	6
4	1	9
5	0	8
6	3	–

(a)

Node	R	r	Node	R	r
0	3	1	6	2	0
1	3	1	7	2	0
2	2	0	8	5	1
3	3	1	9	2	0
4	2	0	10	5	1
5	2	0			

(b)

Fig. 5. R and r both realize the same static schedule.

In general, we can find any down-rotatable set, rotate it down, and reschedule it at the end of the original schedule according to the DAG of the retimed DFG. Assume that s is a schedule of length k with range [1, k]. Let S_i be the set of nodes scheduled in the first i control steps. From property one, we know that S_i is a down-rotatable set for every i in [1 , k]. Only the down-rotatable sets S_i are considered. The number of control steps rotated down in one down rotation is called

the *size* of a down-rotation. We have a new valid schedule s' of the same length k with range $[i+1, i+k]$, i.e.,

$$s'(v) = \begin{cases} s(v)+k, & \text{if } 0 < s(v) <= i \\ s(v), & \text{otherwise.} \end{cases}$$

Notice that s' is a valid schedule for DFG G_{S_i}. Thus, after any down rotation, there always exists a DAG schedule for the retimed DFG, which is at least as short as the original one. The nodes in the last i control steps of s' can be rescheduled together with the partial schedule s' in range [i +1, k] to get a shorter schedule. The experiments on benchmarks, presented in Section VI, show that we are able to converge DAG schedules to the optimal lengths in almost all cases.

In the algorithm description, we use PartialSchedule (G, s, X) to denote a procedure which returns a DAG schedule of G without changing the existent schedule s for nodes in $V - X$. In examples throughout this paper, we use list scheduling for the procedure PartialSchedule(G, s, X) with the number of descendants as the weight function. The procedure DownRotate performs one rotation of size i on a schedule s of DFG G and returns a new schedule of length L as shown at the bottom of the page.

The basic algorithm can handle single-cycle and chained operations. It is implemented in an efficient way. Only nodes in X and the nodes connected to X are involved in computing new weights in list scheduling and precedences. We do not need to construct the DAG of G_X, so a lot of computation is saved. Any incremental scheduling algorithm which does not change the scheduled part can serve as the algorithm for rescheduling.

Section IV will refine the basic algorithm to handle the pipelined functional units and multicycle operations. In Section V, a couple of heuristics are designed to apply a sequence of down-rotations more effectively.

B. Depth of Loop Pipelining

The rotation function R of a node v is incremented by one whenever node v is rotated down, i.e., shifted up, by one iteration, as shown in Fig. IV. The length of prologue and epilogue of a pipeline schedule is proportional to the pipeline depth. Although a sequence of rotations might produce a rotation function with a large difference between $\min_v R(v)$ and $\max_v R(v)$, an efficient retiming algorithm can be applied to reduce that difference by finding a new rotation function.

For a given static schedule realized by a rotation function R, we present an algorithm to find a loop pipeline with a shallow depth by a single-source shortest path algorithm. This algorithm will be used only on the optimal schedules found by our heuristics. For the differential equation example, an optimal schedule in Fig. 5(a) is obtained after seven rotations of size two. We will reduce the depth of the rotation function R accumulated from the sequence of rotations from four to two by finding a new retiming r, as shown in Fig. 5(b). Although G_R and G_r are not equivalent, they realize the same static schedule.

For any static schedule, we use a simple integer linear programming formulation (ILP form) to find a retiming r with smaller $\max_v r(v)$ such that the given static schedule is a DAG schedule of the retimed DFG G_r. From Lemma 1, we can generate an ILP form, which is the dual of a shortest path problem [11] and solvable in time $O(|V|\,|E|)$ [12].

Theorem 2: Let s be a schedule of the DFG G. There exists a legal retiming r such that $s(u)+t(u) \leq s(v)$ for $d_r(u,v) = 0$ if and only if there exists a solution for the following LP form:

$$r(v) - r(u) \leq d(u,v), \quad \text{for every } (u,v) \in E, \text{and}$$
$$r(v) - r(u) \leq d(u,v) - 1$$
$$\text{for every } (u,v) \in E \text{ such that } s(u) + t(u) > s(v).$$

Proof: The inequality $r(v) - r(u) \leq d(u,v)$ ensures that $d_r(u,v) \geq 0$, i.e., r is a legal retiming of G. Since r is a legal retiming, the statement that $s(u)+t(u) \leq s(v)$ if $d_r(u,v) = 0$ is equivalent to the statement if $s(u) + t(u) > s(v)$ then $d_r(u,v) \geq 1$. Since $d_r(u,v) = d(u,v) + r(u) - r(v)$, the inequality $d_r(u,v) \geq 1$ becomes $r(v) - r(u) \leq d(u,v) - 1$. Thus, the theorem is proved. □

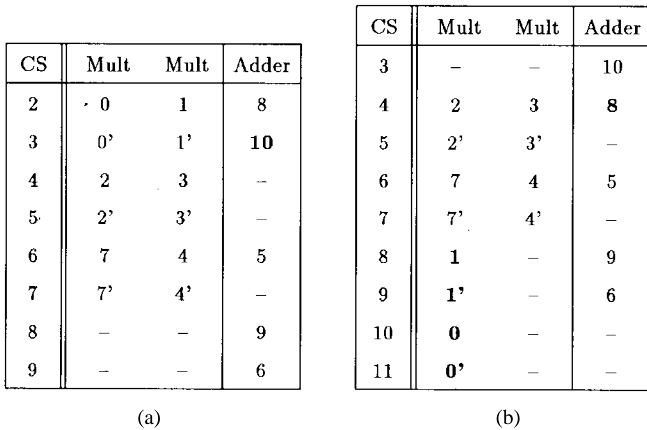
A single-source shortest path algorithm can find a retiming r with small $\max_v r(v)$. We construct a graph $H = (\{v_0\} \cup V, E_H, l_H)$ from the above LP form, where v_0 is a pseudonode not in V. For every inequality $r(v) - r(u) \leq k$ in the LP form, where k is either $d(u,v)$ or $d(u,v) - 1$, we add an edge (u,v) to the edge set E_H with length $l_H(u,v) = k$. For every node v in V, an edge from v_0 to v with length zero is added to E_H.

Lemma 3: If there is a negative cycle in the graph H, there is no solution for the LP form in Theorem 2, and the given static schedule is illegal. Otherwise, the retiming $r(v) = -Sh(v)$ for every $v \in V$ is a solution for the LP form, where $Sh(v)$ is the length of the shortest path from node v_0 to node v in graph H.

Proof: If there is a negative cycle in the graph H, the LP form is not consistent and, thus, have no solutions. Assume that there is no negative cycle in the graph H. From the definition of $Sh(v)$, for every edge (u,v) in E_H, we know $Sh(u) \leq Sh(v) + l_H(u,v)$. By substituting $-r(v)$ into the above inequality, we have $r(v) - r(u) \leq l_H(u,v)$. Thus,

[2] For examples in this paper, we do not shift schedules up after each down rotation so that the readers can compare the schedules before and after rotation easily.

```
DownRotate(G, s, i)
begin
   X ← {v | 1 ≤ s(v) ≤ i};
   Deallocation nodes in X from schedule s;
   Shift s up by i control steps^2;
   s ← PartialSchedule(G_X, s, X);
   Return (s, L, X); /* X: the set of nodes rotated down. */
end
```



CS	Mult	Mult	Adder
2	0	1	8
3	0’	1’	**10**
4	2	3	–
5	2’	3’	–
6	7	4	5
7	7’	4’	–
8	–	–	9
9	–	–	6

(a)

CS	Mult	Mult	Adder
3	–	–	10
4	2	3	**8**
5	2’	3’	–
6	7	4	5
7	7’	4’	–
8	**1**	–	9
9	**1’**	–	6
10	**0**	–	–
11	**0’**	–	–

(b)

Fig. 6. Two rotations of size one for multicycle operations.

$r(v) = -Sh(v)$ is a solution for the LP form in Theorem 2. □

Note that the retiming obtained by this algorithm has the property that $\min_v r(v) = 0$, i.e., r is normalized.

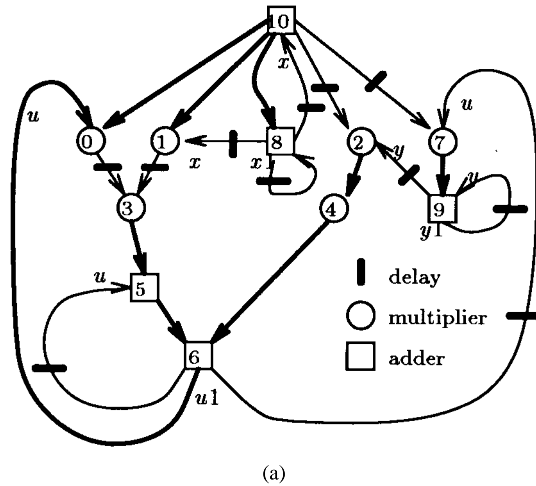
IV. Multicycle Operations and Pipelined Functional Units

In this section, we discuss how to use the basic algorithm to perform down rotations on a schedule when pipelined functional units and multicycle operations are involved. We assume that the computation time of a multicycle operation is an integral multiple of the length of a control step. In our model for a pipelined functional unit, an operation can start execution in every control step, i.e., each stage in the pipeline takes one control step to finish.

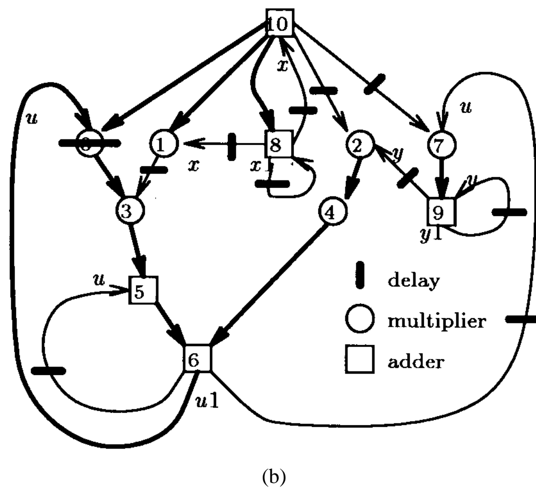
When the starting time of a multicycle operation is in the first i control steps, the operation is rotated down by a rotation of size i. If the multicycle operation does not finish before the ith control step, we still rotate the whole node down. The post-rotation schedule may be longer than the prerotation schedule because of some multicycle operations. The same is true for pipelined functional units. We will apply a technique, called *wrapping*, to overcome this problem.

Consider the differential equation example with the assumptions that a multiplication, an addition, and a control step takes two, one, and one time units, respectively. Fig. 6 shows the schedules after the first and second rotations of size one. Multicycle operations are involved in the second rotation, and the tail of node 0, denoted by 0’, causes the schedule length to increase.

We can *wrap* the tail of node 0 up to the first control step of the schedule and obtain the *wrapped schedule* in Fig. 8(b). There are two conditions to check for a valid wrapping. First, there are spare resources, which is satisfied here. Second, the precedence constraints should be satisfied. If the tail of node 0 is wrapped up, a delay should be pushed half way into node 0, as shown in Fig. 7. The new precedence constraint from node 0’ to node 3 needs to be satisfied. In general, the outgoing edges of a wrapped node with one delay are the new precedence constraints, because they become no-delay edges after the wrapping.

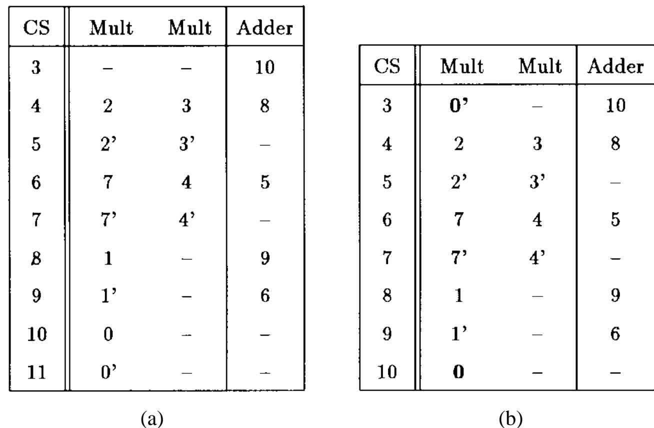


(a)



(b)

Fig. 7. Corresponding retimed graphs after rotations.



CS	Mult	Mult	Adder
3	–	–	10
4	2	3	8
5	2’	3’	–
6	7	4	5
7	7’	4’	–
8	1	–	9
9	1’	–	6
10	0	–	–
11	0’	–	–

(a)

CS	Mult	Mult	Adder
3	**0’**	–	10
4	2	3	8
5	2’	3’	–
6	7	4	5
7	7’	4’	–
8	1	–	9
9	1’	–	6
10	**0**	–	–

(b)

Fig. 8. Wrapping of multicycle operations.

Thus, the schedule length of a DFG with multicycle operations is defined as *the length of the wrapped schedule*. The wrapping only needs to be performed after the last rotation. The rotations in our example still proceed from the unwrapped version of the schedule. After eight rotations of size one from the initial schedule, we get a wrapped schedule of length six.

Sometimes, a wrapped schedule can be easily rotated to be an unwrapped one. A schedule of a loop can be regarded as

a cylinder of instructions, which are repeatedly executed. We can consider any control step i as the first control step of the cylinder by rotating control steps before i without rescheduling them. Therefore, the wrapped schedule in Fig. 8(b) can be rotated into an unwrapped schedule by rotating down control step three with control step four as the first control step. The differences among these schedules are the prologues because they have different rotation functions.

The rotation for nodes implemented as pipelined functional units is similar. The only difference is resource allocation. For a multicycle operation, we have to allocate a resource for every control step during which this operation is executing. While, for a pipelined functional unit, we only need to allocate a resource in the control step in which this operation starts execution. The usage of the remaining stages in the following control steps is automatically available. When precedence constraints are checked, the computation time of a pipelined operation is the number of stages multiplied by the length of a control step.

V. Heuristics

In this section, we propose two heuristics based on the technique of rotation scheduling. As described in Section III-A, any incremental DAG scheduling algorithm can be incorporated with the rotation technique for rescheduling. From an initial schedule, rotations can be repeatedly applied to improve a schedule. At each rotation step, a rotation size has to be identified to decide the amount of changes from the previous schedule. A sequence of a given number of rotations of the same size is called a *rotation phase*. Two simple heuristics are proposed. The first heuristic performs several rotation phases of different sizes individually. The second heuristic performs a sequence of rotation phases of different sizes. Intuitively, the larger the change, the larger the improvement. However, the larger the change, the more time spent on rescheduling. To balance the tradeoffs, the second heuristic starts from a rotation phase of larger size and converges to a rotation phase of size one.

For each rotation phase of *size* i, a given number, denoted by δ, of rotations of size i are performed. After a few rotations, the schedule length might become shorter than the size of rotation; any further rotation of that size is illegal. Therefore, a rotation phase of size i starts with rotations of size i and then decreases the rotation size when the schedule length found is smaller than i.

The procedure of a rotation phase of size i is described as follows. An initial schedule S_{init} and a set Q of current shortest schedules of length L_{opt} are given as inputs. The outputs are the resultant schedule S after all rotations and a set Q of shortest schedules of length L_{opt}. For a schedule of length L, a rotation of size i with $i \geq L$ is illegal. During a phase of size i, when the rotation size is larger than the schedule length L, we divide i by 2 until $i < L$

In the worst case where i is large, each node is rotated at most δ times in a phase. It takes time $O(|V| + |E|)$ to rotate all nodes down once. The time complexity of a rotation phase is $O(\delta\,|E|)$.

In the first simple heuristic, we start every phase from the initial list scheduling of the original DFG. Phases of different sizes are performed according to the length of the initial schedule. The phases in this heuristic are performed independently of each other. Hence, the behavior of this heuristic is more predictable. The larger the value of δ, the better the performance in each phase. The larger the value of ρ, the better the final schedule we obtain. In the algorithm descriptions, we use $\mathsf{FullSchedule}(G)$ to denote a procedure which returns a DAG schedule of G as shown at the bottom of the page.

Since each phase of different size is performed individually, we can study the effect of rotation size on the convergence speed to the optimal schedule length. In general, the convergence speed is faster when the rotation size is large. However, the speed does not increase monotonically with the rotation size. Some irregularities exist. If the rotation size is too small, the corresponding rotation phase may never converge to an optimal schedule length. Therefore, rotation sizes within a certain range should be performed to increase the possibility of reaching an optimal solution. The convergence speed also depends on the amount of resources available. The more resources are allocated, the faster the convergence speed is.

In the second heuristic, each phase uses the final rotation function of the previous phase as an initial retiming function. Since the schedule is improved from phase to phase, the rotation function tends to reflect a better DAG structure for the DFG. These rotation functions give us more faces of the input DFG, and may expose us to chances of better schedules. The initial scheduling of a phase is performed on the retimed DFG. Since rotations of larger size tends to converge to optimal solutions faster, we perform the phases in decreasing order of sizes. This heuristic is described at the bottom of the next page.

The time complexity of the two heuristics are $O(\delta|V||E|)$, assuming that the schedule length is no more than $|V|$. These simple heuristics give very good experimental results to be presented in the next section. In most cases, the two heuristics get the same results. However, the second heuristic gives better schedules in one of the cases. The next section will report the results based on the second heuristic.

```
Heuristic 1(G, δ, ρ)
begin /*  δ : the number of down-rotations in each phase.
          ρ : the range of phases of different sizes. */
      S_init ← FullSchedule(G);
      L_opt ← length of S_init;   Q ← {S_init};
      for i = 1 to ρL do    /* For every phase of size i */
            (L_opt, Q, S, R) ← RotationPhase(S_init, L_opt, Q, G, i, δ);
end
```

VI. Experimental Results

We have experimented with our strategy on several benchmarks, and the results are very promising. All our results are as good as or better than other systems which perform loop pipelining under the same assumptions. In fact, we can prove all the results except one are optimal. Our results are compared against the following three systems: Percolation-based scheduling (PBS) with loop pipelining [15], [17], MARS design system [23], and the scheduler by Lee *et al.* [12]. Cathedral II is not included for comparison because of insufficient data in their paper [8].

We use LB to denote the LB's we derive for different resource constraints, RS to denote our rotation scheduling, and the number in parentheses is the pipeline depth. The figures of other systems appearing in the following tables are adopted from the papers referenced above.

In our experiments, we use a simple list scheduling for both procedures FullSchedule and PartialSchedule with the number of descendants as the weight function. The input DFG's are the original DFG's of the benchmarks. In most cases, the two heuristics presented in Section V get the same results, except for the $2A\ 1M_p$ case of Table II, where the second heuristic gets a better result. The schedules we obtained usually have shallow pipeline depth, like two or three.

It is assumed that the computation time of an adder for one addition is 40 ns, the computation time of a multiplier for a multiplication is 80 ns, and the clock cycle period for a control step is 50 ns with 10 ns for the latch time of buffers. The pipelined multiplier, denoted by M_p, is assumed to consist of 2 stages, each of which takes no more than 40 ns. The numbers in the tables are the number of control steps.

The experiments are done on the five benchmarks in Table I. The Critical Path (CP) equals the minimum length of schedules without loop pipelining. The Iteration Bound (IB) is the theoretical LB on the schedule length, which is the ceiling of the maximum time-to-delay ratio among all cycles in the DFG [19].

The results for the elliptic filters[3] are shown in Table II, and the rest of the results are shown in Table III. Our results compare favorably with those of other systems. Every experiment is finished within seconds on a DEC 5000 workstation in C programming language. The experiments for elliptic filters are performed in 2.5 s, those for the other four benchmarks in less than 1 s. For the elliptic filters, the number of optimal schedules found ranges from 15 to 35, depending on the availability of resources. The first optimal schedule is usually found within 1 s.

In addition, we compare our results with the theoretical lower bounds (LB's) we derive in [4]. Larger LB's on the schedule lengths are obtained when we have more strict resource constraints. For the elliptic filter, we achieve the derived LB's, except in the case of $2A\ 1M$. In Table III, we always meet the LB's in every resource requirements for the other four benchmarks.

[3] As pointed out by [16], there are errors in the DFG of [9]. We derived the correct DFG from the signal flow graph description of the elliptic filter in [9].

TABLE I
Characteristics of the Benchmarks

Benchmark	#Mults	#Adds	CP	IB
5-th Order Elliptic Filter	8	26	17	16
Differential Equation	6	5	7	6
4-stage Lattice Filter	15	11	10	2
All-pole Lattice Filter	4	11	16	8
2-cascaded Biquad Filter	8	8	7	4

CP: Critical Path

IB: Iteration Bound

TABLE II
Results for the Elliptic Filters

Resources	**LB**	PBS	MARS	Lee *et al.*	**RS**
Nonpipelined Multipliers					
$3A\ 3M$	**16**	16	n/a	16	16 (2)
$3A\ 2M$	**16**	17	n/a	16	16 (2)
$2A\ 2M$	**17**	17	n/a	17	17 (2)
$2A\ 1M$	**17**	20	n/a	19	19 (2)
Pipelined Multipliers					
$3A\ 2M_p$	**16**	16	n/a	16	16 (2)
$3A\ 1M_p$	**16**	16	16	16	16 (2)
$2A\ 1M_p$	**17**	18	17	17	17 (2)

```
Heuristic 2(G, δ, ρ)
begin /* δ, ρ : as in Heuristic 1. */
    S ← FullSchedule(G);
    L_opt ← length of S;   Q ← {S};   R ← ∅;
    /* iterative compaction */
    for i = ρL to 1 by −1 do begin    /* size-i phase */
        (L_opt, Q, S, R) ← RotationPhase(S, L_opt, Q, G, i, δ);
        /* Find a new initial schedule for the next phase */
        S ← FullSchedule(G_R);   L ← length of S;
        if L < L_opt then   L_opt ← L; Q ← {S}; end;
        else if L = L_opt then   Q ← Q ∪ {S};
    end
end
```

TABLE III
EXPERIMENTAL RESULTS FOR THE OTHER FOUR BENCHMARKS

Resources	Pipelined Multipliers LB	Pipelined Multipliers MARS	Pipelined Multipliers **RS**	Nonpipelined Multipliers Resources	Nonpipelined Multipliers LB	Nonpipelined Multipliers **RS**
Differential Equation						
$1A\ 1M_p$	6	n/a	6(2)	$1A\ 2M$	6	6(2)
				$1A\ 1M$	12	12(2)
The 4-stage Lattice Filter						
$6A\ 8M_p$	2	2	2(6)	$6A\ 15M$	2	2(5)
$4A\ 5M_p$	3	n/a	3(4)	$4A\ 10M$	3	3(5)
$3A\ 4M_p$	4	n/a	4(3)	$3A\ 8M$	4	4(3)
$3A\ 3M_p$	5	n/a	5(2)	$3A\ 6M$	5	5(4)
$2A\ 3M_p$	6	n/a	6(2)	$2A\ 5M$	6	6(2)
$2A\ 2M_p$	8	n/a	8(2)	$2A\ 4M$	8	8(2)
All-pole Lattice Filter						
$3A\ 2M_p$	8	8	8(3)	$3A\ 2M$	8	8(3)
$2A\ 2M_p$	9	n/a	9(2)	$2A\ 2M$	9	9(2)
$2A\ 1M_p$	9	n/a	9(2)	$2A\ 1M$	10	10(2)
$1A\ 1M_p$	11	n/a	11(2)	$1A\ 1M$	11	11(2)
The 2-cascaded Biquad Filter						
$2A\ 2M_p$	4	4	4(2)	$2A\ 4M$	4	4(2)
$2A\ 1M_p$	8	n/a	8(2)	$2A\ 3M$	6	6(2)
$1A\ 2M_p$	8	n/a	8(2)	$1A\ 2M$	8	8(2)
$1A\ 1M_p$	8	n/a	8(2)	$1A\ 1M$	16	16(2)

VII. COMPARISON WITH PREVIOUS WORK

This section surveys and compares our approach with previous work from the literature of high-level synthesis for very large scale integration (VLSI) design and parallel compilers for VLIW machines. The papers [5], [18] on data-flow graph transformation consider the combination of retiming and algebraic transformations without using any particular scheduling algorithm. Loop pipelining algorithms for high-level synthesis include Lee, Wu, Gajski, and Lin [12], Wang and Parhi [23], Goossens, Vandewalle, and De Man [8], Potasman, Lis, Nicolau, and Gajski [15], [17]. Ebcioglu and Nakatani [6] and M. Lam [10] are two papers on parallel compilers for VLIW machines. We believe that our loop pipelining framework can be applied to this field.

The scheduling algorithms in [12] and [23] are length-constrained to minimize the amount of resources. The algorithms in [8] and [10] are both length-constrained and resource-constrained. In order to optimization execution rate, a chosen schedule length has to be updated iteratively. The percolation scheduling, developed by Nicolau [14] and used in both Ebcioglu and Nakatani [6] and Potasman *et al.* [15], [17], minimizes schedule length through a sequence of local transformations under resource constraints.

Potkonjak and Rabaey [18] and Chao [5] have proposed transformation-based algorithms to combine retiming and other algebraic transformation, such as associativity and distributivity. In our context, algebraic transformation and retiming can be considered in a preprocessing step to generate optimized data-flow graphs. Our loop pipelining algorithm can be applied on the transformed data-flow graphs to obtain a real schedule with possible modifications on retiming.

The algorithms by Lee *et al.* [12] and in MARS [23] are designed for time-constrained scheduling, while we consider resource-constrained scheduling to maximize the execution rate. Therefore, the approaches are quite different. In their algorithm, schedules are first generated to satisfy time constraints, and operations are rescheduled selectively to minimize the amount of resources. The MARS system [23] first finds all cycles in the DFG and computes the loop bound. A set of cycle sections is derived to cover all cycles. Resource conflicts are resolved after the cycle sections are scheduled individually. The nodes not belonging to any cycle are scheduled at the last step. Retiming is performed implicitly in MARS. Recently, the functional pipelining algorithm by Lee *et al.* [12] uses a priority function, called *variability*, for rescheduling operations in an initial as-soon-as-possible pipeline schedule to resolve resource conflicts.

In Cathedral II [8], a DFG is retimed to meet an estimated schedule length without resource constraints, and, then, another graph is constructed from the DFG and retiming function to reschedule the loop entirely under resource constraints. Iteratively, the estimated schedule length is decreased one by one from an upper bound obtained from list scheduling. Usual retiming algorithms only find one retimed graph for a given schedule length without considering any resource constraints. However, there are usually a great number of retimed graphs with the same schedule length. Some are good for certain resource requirements, but some are not. Our step-by-step rotation approach enables us to find retimed graphs under resource constraints.

Percolation-based scheduling (PBS) [15], [17] unfolds (unwinds) the loop incrementally to find a repeating pattern in the schedule of the unfolded loop without resource constraints and then schedules it with resources. The size of the pipeline schedule can not be predicted until several incremental unfoldings are applied. The unfolding of loops is considered in the front end of our system to generate a data-flow graph with high execution rate [2], [3], where the size of repeating pattern can be controlled. Also, only one repeating pattern is found in PBS without resource constraints. Many repeating patterns can be generated from our rotation technique, and each is generated with respect to resource requirements.

A software pipelining algorithm under resource constraints is proposed by Lam to design optimizing compilers for the Warp machine [10]. A data/control flow graph is first analyzed to find connected components, and each connected component is scheduled individually. The graph is reduced to an acyclic graph by representing each component as a single vertex. In connected-component scheduling, the range within which each operation can be legally scheduled is updated with respect to the current partial schedule. Thus, the legal ranges of

operations have to be updated after one more operation is scheduled. In order to compute such ranges, a given schedule length has to be provided. Our approach schedules the entire graph uniformly. Operations from different connected components are scheduled under joint resource constraints. This gives the opportunities of sharing resources among connected components and exploring alternative schedules for coupled connected components.

The observation that moving the first instruction to the end of loop body can achieve the effect of loop pipelining has been made by Ebcioglu and Nakatani [6] in the context of parallelizing compilers for a VLIW machine. An enhanced percolation scheduling algorithm is used to reschedule the entire loop body after each move. Our algorithm moves a set of instructions/operations legally and only reschedule these operations. All these features are based on the framework we developed to relate retiming with loop pipelining. Therefore, each rotation step of our algorithm is efficient and simple. From the retiming function which realizes a pipeline schedule, the correlation between the new loop body and the old loop body is clear.

VIII. Conclusion

In this paper, we presented a theoretical foundation and experimental results for a new transformation, called *rotation*, for data-flow graphs. This technique is simple and easy to implement. It can be incorporated with any incremental DAG scheduling algorithm without pipelining.

We showed the effectiveness of the technique of rotation scheduling under various resource constraints in our experiments. In addition, through a sequence of rotations, many optimal schedules can be found, which expose more chances of optimization for the following stages of high-level synthesis, e.g., connection binding, allocation, or data-path generation.

Rotation is a generic technique for loop pipelining. Although strictly speaking, it is a restricted form of retiming, the concept of rotation is easier to manipulate in an optimization process. We have demonstrated its effectiveness by very simple heuristics, based on the procedure RotationPhase. There are a class of heuristics to be developed by varying the parameters δ, ρ.

An approach is proposed to reduce the depth of loop pipelining after a large number of rotations. Since our algorithm focuses on improving the schedule length of the repeating part, it may not work well for a loop with a small number of iterations, though most DSP applications require a large number of iterations, such as recursive filters.

The rotation technique can be extended to handle nested loop pipelining. We schedule loops from inside out. The innermost loop is scheduled and pipelined first and partitioned into the prologue, static schedule, and epilogue. When rotations are applied on the outer loop, the static-schedule part is treated as a compound node, which occupies several functional units and takes several control steps to complete. The prologue, epilogue, and the compound node are also represented in the data-flow graph of the outer loop. Therefore, the schedules of the inner and outer loops blends together. Similar approaches have been used in [6] and [10].

As described in Section III-A, any incremental scheduling algorithm which does not change the scheduled part can serve as the algorithm for rescheduling. The proposed framework of loop pipelining can be incorporated with other incremental scheduling algorithms with capabilities to handle other forms of resource constraints, such as interconnection costs [22] and register costs or extensions to more complicated models, such as conditionals [20].

References

[1] L.-F. Chao and E. H.-M. Sha, "Retiming and unfolding data-flow graphs," in *Proc. Int. Conf. Parallel Processing*, St. Charles, IL, vol. 2, Aug. 1992, pp. 33–40.

[2] ——, "Scheduling data-flow graphs via retiming and unfolding," *IEEE Trans. Parallel Distrib. Syst.*, to be published.

[3] ——, "Static scheduling for synthesis of DSP algorithms on various models," *J. VLSI Signal Processing*, vol. 10, pp. 207–223, 1995.

[4] L.-F. Chao, "Scheduling and behavioral transformations for parallel systems," Dept. Comput. Sci., Princeton Univ., Princeton, NJ, Tech. Rep. CS-TR-430-93, July 1993.

[5] ——, "Optimizing cyclic data-flow graphs via associativity," in *Proc. Great Lakes Symp. VLSI*, Mar. 1994, pp. 6–10.

[6] K. Ebcioglu and T. Nakatani, "A new compilation technique for parallelizing loops with unpredictable branches on a VLIW architecture," in *Languages and Compilers for Parallel Computing*. Cambridge, MA: MIT Press, 1990, pp. 213–229.

[7] E. M. Girczyc, "Loop winding—A data flow approach to functional pipelining," in *Proc. Int. Symp. Circuits Syst.*, May 1987, pp. 382–385.

[8] G. Goossens, J. Vandewalle, and H. De Man, "Loop optimization in register-transfer scheduling for DSP-systems," in *Proc. ACM/IEEE Des. Automat. Conf.*, 1989, pp. 826–831.

[9] S. Y. Kung, H. J. Whitehouse, and T. Kailath, *VLSI and Modern Signal Processing* (Information and Systems Sciences Series). Englewood Cliffs, NJ: Prentice-Hall, 1985, pp. 258–264.

[10] M. Lan, "Software pipelining: An effective scheduling technique for VLIW machines," in *Proc. ACM SIGPLAN Conf. Programming Languages Des. Implement.*, Atlanta, GA, June 1988, pp. 318–328.

[11] E. L. Lawler, *Combinatorial Optimization: Networks and Matroids*. New York: Holt, Rinehart, and Winston, 1976.

[12] T.-F. Lee, A. C.-H. Wu, D. D. Gajski, and Y.-L. Lin, "An effective methodology for functional pipelining, in *Proc. Int. Conf. Comput.-Aided Des.*, Dec. 1992, pp. 230–233.

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

[14] A. Nicolau, "Percolation scheduling: A parallel compilation technique," Dept. Comput. Sci., Cornell Univ., Ithaca, NY, Tech. Rep. TR 85-678, 1985.

[15] A. Nicolau and R. Potasman, "Incremental tree height reduction for high level synthesis," in *Proc. ACM/IEEE Des. Automat. Conf.*, 1991, pp. 770–774.

[16] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of ASIC's," *IEEE Trans. Computer-Aided Design*, vol. 8, pp. 661–679, June 1989.

[17] R. Potasman, J. Lis, A. Nicolau, and D. Gajski, "Percolation-based scheduling," in *Proc. ACM/IEEE Des. Automat. Conf.*, 1990, pp. 444–449.

[18] M. Potkonjakk and J. Rabacy, "Optimizing resource utilization using transformations," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 277–292, Mar. 1994.

[19] M. Renfors and Y. Neuvo, "The maximum sampling rate of digital filters under hardware speed constraints," *IEEE Trans. Circuits Syst.*, vol. CAS-28, pp. 196–202, Mar. 1981.

[20] J. Siddhiwala and L.-F. Chao, "Scheduling conditional data-flow graphs with resource sharing," in *Proc. Great Lakes Symp. VLSI*, Mar. 1995, pp. 94–97.

[21] R. E. Tarjan, *Data Structures and Network Algorithms*. Philadelphia, PA: SIAM, 1983.

[22] S. Tongsima, N. Passos, and E. H.-M. Sha, "Communication sensitive rotation scheduling," in *Proc. Int. Conf. Comput. Des.*, Oct. 1994, pp. 150–153.

[23] C.-Y. Wang and K. K. Parhi, "High level DSP synthesis using the MARS design system," in *Proc. Int. Symp. Circuits Syst.*, 1992, pp. 164–167.

Liang-Fang Chao (S'91–M'93) received the B.S. and M.S. degrees in computer engineering from National Chiao Tung University, Taiwan, in 1986 and 1988, respectively, and the M.A. and Ph.D. degrees in computer science from Princeton University, Princeton, NJ, in 1991 and 1993, respectively.

From July 1988 to May 1989, she was an Assistant Research Engineer at the Telecommunication Laboratories, Taiwan. In August of 1993, she joined the Department of Electrical and Computer Engineering, Iowa State University, Ames, as an Assistant Professor. Her research interests include computer-aided design of VLSI systems, special-purpose architecture design, and parallel and distributed computing.

Dr. Chao organized the 1996 Great Lakes Symposium on VLSI as a Technical Program Co-Chair. She is a recipient of the 1994 Research Initiation Award.

Andrea S. LaPaugh (M'82) received the A.B. degree in physics from Cornell University, Ithaca, NY, in 1974 and the E.E. and Ph.D. degrees in electrical engineering and computer science from Massachusetts Institute of Technology, Cambridge, in 1977 and 1980, respectively.

She is currently a Professor of Computer Science at Proinceton University, Princeton, NJ, where she has been on the faculty since 1981. Prior to that, she was a Visiting Assistant Professor with the Department of Computer Science, Brown University, Providence, RI. Her research interests include several areas within computer-aided design of digital systems, including physical design and high-level architecture design. Her work applies theoretical design and analysis techniques for combinatorial algorithms to computer-aided design problems.

Dr. LaPaugh is a member of the ACM (including SIGACT, SIGARCH, and SIGDA).

Edwin Hsing-Mean Sha (S'89–M'92) received the B.S.E. degree in computer science and information engineering from National Taiwan University, Taipei, Taiwan, in 1986 and the M.A. and Ph.D. degrees from the Department of Computer Science, Princeton University, Princeton, NJ, in 1991 and 1992, respectively.

Since August 1992, he has been with the University of Notre Dame, Notre Dame, IN, where he is an Assistant Professor, and since August 1995, he has been Associate Chairman for Graduate Studies of the Department of Computer Science and Engineering. His research areas include computer-aided design for application-specific parallel and pipelined architectures, data scheduling and partitioning for parallel systems, loop transformations and parallelizations, software tools for parallel and distributed systems, high-level synthesis in VLSI, fault-tolerant computing, VLSI processor arrays, and hardware and software codesign.

Dr. Sha received the ORAU Junior Faculty Enhancement Award in 1994 and the National Science Foundation Career Award in 1995. He served as the Technical Committee Co-Chair for the 1994 IEEE Great Lakes Symposium on VLSI and was on the program committees for several IEEE international conferences.