PLS: A Scheduler for Pipeline Synthesis

Cheng-Tsung Hwang, Yu-Chin Hsu, *Member, IEEE,* and Youn-Long Lin, *Member, IEEE*

***Abstract*—Pipelining is an effective method to optimize the execution of a loop, especially for digital signal processing (DSP) applications where data enter a circuit regularly. Although throughput and delay are two important optimization criteria, previous work emphasizes mainly on the throughput. We show that the delay time of executing an iteration of a loop has a strong relationship with the cost of the registers and the controller. By minimizing the delay, we could have more silicon area to allocate additional resources which in turn will increase throughput. We iteratively use a forward scheduling and a backward scheduling to achieve this purpose. The algorithm can be used to pipeline a loop with or without loop carried dependencies. Real examples are used to illustrate the proposed method. Experiments on benchmark examples show that the new approach gains a considerable improvement over those by previous approaches.**

I. Introduction

THE AUTOMATIC synthesis of a digital system from a behavioral description to a structure design requires several tasks to be performed. Among them, operation scheduling and hardware allocation are the most important. Scheduling assigns each operation a specific control step; while allocation binds every behavioral object a hardware resource. Usually, the performance of a digital system is constrained by both the available hardware resources and the algorithm structure (i.e., precedence relationship). Thanks to the progress in VLSI processing techniques, using massive hardware becomes practical. Consequently, it is very important to explore the concurrency of the algorithm in order to implement a high-throughput system.

Highly concurrent implementations can be obtained by executing several operations within one iteration in parallel and/or by overlapping the execution of consecutive iterations. During the past decade, the concurrency within one iteration has been extensively studied [1]. Recently, more effort has been devoted to explore concurrency beyond the boundary of a single iteration. Loop unrolling and software pipelining are two well-known methods [2]. Loop unrolling is done by replicating a loop multiple times, then scheduling the unrolled loop. Software pipelining is a technique for organizing loops such that the software-pipelined code contains different iterations in the original code segment.

Manuscript received September 16, 1991; revised February 12, 1992. This paper was recommended by Associate Editor A. Parker.

C.-T. Hwang and Y.-C. Hsu are with the Computer Science Department, University of California, Riverside, CA.

Y.-L. Lin is with Tsing Hua University, Hsin-Chu, Taiwan.

IEEE Log Number 9208518.

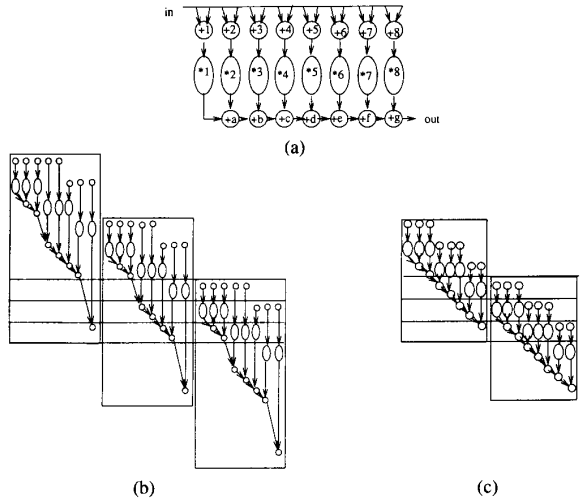

Fig. 1. 16-point digital FIR filter with 5 adders and 3 multipliers. The latency of the design is 3 and clock cycle time is 100 ns. (a) *DFG* representation. (b) Pipeline schedule 1. (c) Pipeline schedule 2.

In some DSP applications (e.g., filters), data enter a circuit regularly. Loop unrolling cannot be applied in such an application because the operations of the unrolled iterations will be scheduled in an irregular way. Pipelining, on the other hand, is very efficient for this purpose. An example of pipelining a loop is given in Fig. 1, where several consecutive iterations are executed simultaneously. An iteration is initiated every three steps (which is termed *latency*) and each iteration in Figs. 1(b) and 1(c) takes nine and six steps to finish, respectively (which is termed *delay*). In general, there may exist data dependencies among operations across different iterations (called *loop carried dependency*, abbreviated as LCD). A feasible pipeline schedule has to satisfy all the data dependencies.

Most of the pipeline synthesis algorithms emphasize the optimization of the throughput. We observe that delay time is also very important for pipeline design. Normally the shorter the delay is, the less the overhead of the control unit and fewer registers are required for the temporary variables. Our approach hence minimizes both the latency and the delay. The remainder of the paper is organized as follows. Section II reviews the related work on pipeline synthesis. In Section III, some fundamental concepts for pipeline design are discussed. The proposed algorithm is presented in Section IV. We first describe an algorithm for pipelining loops without LCD's, then extend it to han-

dle loops with LCD's. In Section V, we show the experimental results. Finally, concluding remarks are made in Section VI.

II. Related Work and Motivation

A pipeline synthesis problem can be constrained either by resource or time, or a combination of both. A resource constraint limits the area of a chip or the available number of function units of each type. A time constraint specifies the required throughput and turn around time. Given the available hardware, the objective of a resource constrained scheduler is to find a schedule with maximum performance. Given the constraint on the throughput and turn around time, the objective of a time constrained scheduler, on the other hand, is to find a schedule consuming minimum hardware.

Sehwa [3] is the first synthesis program in the literature featuring functional pipelining scheduling. In that paper, the theoretical foundation has been laid down for pipelining a loop without loop-carried dependencies. It has been shown that given a constraint on the number of resources, a pipelined data path can be implemented with minimum latency. Since latency is fixed, the objective is to minimize delay. Sehwa tackles this problem using a modified list scheduling and an exhaustive search algorithm with the aid of a resource allocation table.

HAL [4] performs a time-constrained, functional pipelining scheduling using the *force directed* method. This method has been adapted and modified in [5].

The idea of *loop winding* has been presented in the Elf [6] system. A loop iteration is partitioned horizontally into several pieces, which are then wound in parallel to form a shorter loop to achieve a higher throughput. A user-specified throughput is used to determine the length of the shorter loop. The percolation based scheduling (PBS) [7] deals with the loop winding problem by starting with an optimal schedule (OPT) which is obtained without considering resource constraints. Then a heuristic is used to adjust the schedule to meet the resource constraints. The OPT can be found by the approach in [8] where a loop is incrementally unwound and its operations are allowed to migrate upwards until a pattern emerges. This method guarantees that a shortest partition can be found if unlimited resources are provided. Spaid [9] finds a maximally parallel pattern using a linear programming formulation.

ATOMICS [10] performs loop optimization starting with an unfolded loop structure. After estimating a latency, inter-iteration precedences are projected onto the precedence constraints of zero degree. Operations which cannot be scheduled within the latency are moved (folded) to the next iteration. When a solution is found, the latency is decreased and the folding is applied again to the folded loop structure. The optimization process terminates after it fails for a number of times.

Among the above approaches, only Sehwa tries to minimize the delay time. Although list scheduling generates acceptable results for non-pipelined synthesis, a direct application to pipelined design may not be adequate as demonstrated by the example in Fig. 1. Different heuristics are needed to meet the new requirements. This motivates us to develop a new algorithm to solve the problem. The important features of the proposed approach as compared to those of the previous approaches are:

1) Unlike Elf, Spaid and PBS which perform partitioning before scheduling, our algorithm *dynamically partitions* the *DFG* during scheduling.
2) Different from that in Sehwa, a scheduled operation can be *rescheduled* to accommodate the incoming ready operations. Also, the proposed method can effectively pipeline a loop with loop carried dependencies.

III. Preliminaries

3.1. Input Representation

The input to the algorithm is a data flow graph $DFG(V, E)$, where V is the set of operations and E is the set of data precedences. An operation is denoted as o_i. An instance of o_i at iteration i_A is denoted as o_i @ i_A. A data precedence relationship between two operations o_i and o_j is denoted as $o_i \rightarrow o_j$. Each edge e is associated with a *degree* d_e which represents the value generated by o_i will be used by o_j at d_e iterations later and a *weight* w_i which specifies the minimum distance between o_i and o_j is w_i. A node is a *source* (*sink*) node if it has no predecessors (successors) in the same iteration.

3.2. The Lower Bound of Latency

To find the minimum latency in the presence of loop carried precedences is an *NP-complete* problem [11]. However, a trivial lower bound on the achievable latency $L_{\min}$ can be obtained by considering the resource constraints and analyzing the precedence constraints.

Let N_k be the number of operations of an iteration which must be performed by a type k function unit and M_k be the number of available function units of type k. The lower bound of latency due to the type k function unit is $\lceil N_k/M_k \rceil$, and the lower bound of latency for a *DFG* is $\max_{1 \le k \le m} \lceil N_k/M_k \rceil$. For example, if a *DFG* contains 5 additions and 6 subtractions, the lower bound of latency, given 3 ALU's, is $\lceil (5 + 6)/3 \rceil = 4$.

In a pipeline schedule, every operation repeats itself with a fixed interval which is the latency. Suppose that there is a *dependency cycle* o_1 @ 1 $\rightarrow$ o_2 @ 1 $\rightarrow \cdots \rightarrow$ o_c @ 1 $\rightarrow$ o_1 @ 2. The cycle will force the repeat of o_1 (o_1 @ 1 and o_1 @ 2) with an interval which is at least $\Sigma_{1 \le i \le c} w_i$, where w_i is the execution delay of o_i. Therefore, the lower bound of latency due to this cycle is $\Sigma_{1 \le i \le c} w_i$. Let *loops* denote the set of all cycles in a *DFG* and $loop_l$ be one of the cycles. Suppose that the total execution delay of all the operations in $loop_l$ is Γ_l (Γ_l =

$\Sigma_{o_i \in loop_l} w_i$) and the total loop degree of the edges in $loop_l$ is D_l ($D_l = \Sigma_{e \in \text{loop}_l} d_e$, where d_e is the degree of the edge e). The lower bound of latency due to precedence constraints is $\max_{loop_l \in loops} \lceil \Gamma_l / D_l \rceil$.

3.3. The Delay of a Pipeline Schedule

It is essential to define input and output of a *DFG* before we can define the delay of a schedule. We say that a node is an *input* node if it takes signals from outside environment and is an *output* node if it produces signal for outside environment. In a pipeline design, several tasks are executed in a pipelined fashion. We define the delay of a task as the duration from the submission of input data until the outcome is available at the output. A task may have multiple input and output signals. It is reasonable to assume that all the input signals are ready before a task can start and a task is completed when all the output signals are produced.

In the presence of loop carried dependencies, the output signals at iteration i_A may depend on the input at any earlier iteration i_B, where $i_B \leq i_A$. However, assume that the last output of the first iteration is produced at time D, the same output of successive iterations will be produced periodically at time $D + L$, $D + 2L$, $D + 3L$, $\cdots$, etc.; i.e., the same output of the Ith iteration will be produced at time $D + (I - 1) \times L$. Therefore, if the delay of the first iteration is minimized, then the delay of successive iterations will be minimized as well. Thus, we define the delay of a pipeline schedule (with or without LCD's) as the duration from the first input node to the last output node *in an iteration*. In order to minimize delay, an output node should be scheduled earlier than an intermediate node whenever it is possible.

IV. The Proposed Method

In this section, we will first describe an algorithm for pipelining a loop without loop carried dependencies, and then extend it to that with loop carried dependencies.

4.1. Pipelining a Loop Without LCD's

Given a data flow graph and resource constraints, we first determine a minimum achievable latency L using the method described in Section 3.2. Then, a pipeline-schedule iteration is performed which iteratively selects and assigns the operations of *DFG* into the pipelined loop body. We call the graph induced by those not yet scheduled operations the *remaining graph* (denoted as DFG_R) and the graph induced by those scheduled operations is called the *scheduled graph* (denoted as DFG_S). During each pass of the algorithm, a set of operations is selected from DFG_R. The subgraph induced by the scheduled operations during pass I is called DFG_I. DFG_S is constructed iteratively by adding the operations just selected and the operations projected by the back edges of the scheduled operations. The algorithm terminates when DFG_R becomes empty.

Initially, DFG_S is empty and DFG_R equals *DFG*. A pipeline-schedule iteration consists of two phases: *forward* scheduling and *backward* scheduling. During forward scheduling, operations are scheduled into the execution window from the first to the last step. The schedule of forward scheduling is modified by a backward scheduling before we move to the next pass. The motivation to modify a schedule by a backward scheduling is that a backward schedule leaves more resources available at the earlier steps. Consequently, it is more likely to assign the incoming operations at the earlier steps during the forward scheduling of the next iteration. Our algorithm fully utilizes the properties of both forward and backward schedulings. A pseudo-code description of the algorithm is shown in Fig. 2. The forward scheduling and backward scheduling are explained in more details.

Forward Scheduling: Forward scheduling proceeds from the first step to the last step. During the scheduling of C_{step}, ready operations may come from both DFG_R (incoming operations) or DFG_S (previously scheduled operations). We schedule incoming operations first for the following reasons: (1) suppose an operation o_i is the last unscheduled node, by scheduling o_i first, the task will be finished with a delay which is shorter than that by scheduling those in DFG_S first; (2) if o_i is the only predecessor of an output node o_j, by scheduling o_i first, o_j becomes ready and can be scheduled at the next step; (3) even the schedule may not be able to finish within this pass, by scheduling incoming operations first, more incoming operations can be included into DFG_S during this pass. Thus, the task can be expected to finish earlier.

After incoming operations are scheduled, operations in DFG_S can be pulled up (which were pulled down by backward scheduling) to C_{step} (if there are resources left) which in turn will leave more resources at later steps that can be used by incoming operations. The above strategy is achieved by the priority function

$$P_f(i) = \begin{cases} l_i, & o_i \in DFG_S \\ A + l_i, & o_i \in DFG_R \end{cases}$$

where l_i is the path length from o_i to the sink nodes in its containing subgraph and A is a sufficiently large integer to make sure the incoming operations get a higher priority.

Backward Scheduling: Backward scheduling proceeds from the last step to the first step. It *pulls down* the previously scheduled operations as much as possible so that the distribution of operations at the earlier steps becomes sparse. The priority function used here is

$$P_b(i) = s_i, \qquad o_i \in DFG_S$$

where s_i is the path length from o_i to the source nodes in its containing subgraph.

As the scheduling proceeds, more operations in the DFG_R are scheduled into DFG_S. They will compete with each other for resources. To stabilize the algorithm, we maintain for each operation a pair of bounds $[S_{\text{low}}, S_{\text{high}}]$.

ALGORITHM PipeLine_Scheduling
Input: DFG, Resources;
Output: a pipelined schedule with latency L and delay D;

(1) Calculate minimum latency L.
$I = 0$, $DFG_S = \emptyset$, $DFG_R = DFG$
(2) **while** ($DFG_R \neq \emptyset$) **do**
$I = I + 1$
(2.1) **for** C_{step} = 1 to L /* Forward_Scheduling */
• Schedule urgent operations
• Schedule incoming operations
• Schedule operations in DFG_S
(2.2) $DFG_S = DFG_S \cup DFG_I$
$DFG_R = DFG_R - DFG_I$

if ($DFG_R = \emptyset$) **break;**

(2.3) **for** $C_{step} = L$ to 1 /* Backward_Scheduling */
• Schedule urgent operations
• Schedule operations in DFG_S
End while

Fig. 2. A pseudocode description of the algorithm for pipelining a loop without loop carried dependencies.

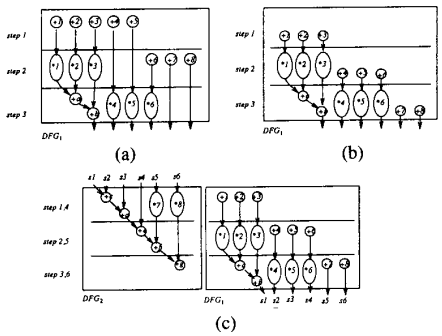

Fig. 3. Example to illustrate the proposed method. (a) After forward scheduling of pass 1. (b) After backward scheduling of pass 1. (c) The final schedule (DFG_S).

S_{high}, which is determined by the forward scheduling, is the *earliest step* into which an operation can be scheduled; while S_{low}, which is determined by the backward scheduling, is the *latest step* into which an operation must be scheduled. Since forward scheduling assigns an operation to the earliest possible control step T_f; while backward scheduling assigns an operation to the latest control step T_b. It is unlikely for an operation to be scheduled later than T_b during forward scheduling. Similarly, it is unlikely that an operation to remain unscheduled at T_f during backward scheduling. Therefore, we define an operation as *urgent* if its S_{low} equals current step C_{step} during forward scheduling or S_{high} equals C_{step} during backward scheduling. Urgent operations should be scheduled immediately to guarantee the algorithm be *stable*. Thus, the scheduled operations are iteratively moved up and down within $[S_{high}, S_{low}]$ to accommodate incoming operations. The range of movement becomes smaller and smaller as scheduling proceeds.

Example: Let us illustrate the idea using the *DFG* depicted in Fig. 1(a). During the first pass of the scheduling, the forward scheduling will put as many operations as possible into DFG_1 (Fig. 3(a)). Before we move on to the second pass, the operations just scheduled are pulled down as much as possible by a backward scheduling (Fig. 3(b)). At the beginning of the second forward scheduling, +1, +2 and +3 are scheduled first because they are urgent. Then +c, +d, *7 and *8 of DFG_R are scheduled into step 1. Because no adders are available for control step 1, no addition in DFG_1 can be pulled up. We proceed to control step 2. Here, *1, *2, *3, +4, +5, +6 (urgent operations) are assigned first, then +e, +f (incoming operations) are scheduled. Finally, urgent operations, +a, +b, *4, *5, *6, +7, +8 and incoming operation +g are assigned into control step 3. The algorithm finishes with a schedule of delay 6, which is optimum (Fig. 3(c)). Notice it takes nine steps to complete the pipeline schedule by just using the forward feasible scheduling.

Complexity: By keeping the ready operations in a priority queue, the operation with the highest priority can be retrieved in $O(\log n)$ time, where n is the number of operations in the *DFG*. Forward scheduling and backward scheduling can be finished in $O(n \log n)$. Since the number of pipeline-schedule iterations is usually small, the complexity of the algorithm is $O(n \log n)$.

4.2. Pipelining a Loop With LCD's

Before we introduce the algorithm for pipelining a loop with LCD's, let us start with the example in Fig. 4 which illustrates how a 1-degree data dependency is taken into account.

In a pipeline schedule, if o_A @ 1 is performed at time T_A, then o_A @ 2 will be performed at time $T_A + L$. Therefore, given a loop carried data dependency of 1-degree $o_i \xrightarrow{d=1} o_j$ (i.e., o_i @ 1 $\rightarrow$ o_j @ 2 $\rightarrow$ o_j @ 3, $\cdots$, etc.), the constraint $T_i \leq T_j + L$ should be obeyed. Consider the scheduling of o_i and o_j in Fig. 4(b), 4(c) and 4(d). In Fig. 4(b), both o_i and o_j are assigned to DFG_1 (the o_i and o_j in DFG_1 of the execution window are o_i @ 2 and o_j @ 2 respectively). Since $|T_i - T_j| \leq L - 1$, the correctness of the constraint $T_i \leq T_j + L$ is implied. Therefore, we don't have to specify the precedence $o_i \rightarrow o_j$ in DFG_S. In Fig. 4(c), o_j is assigned to DFG_1 and o_i is assigned to DFG_2. The o_i and o_j in the execution window are actually o_i @ 1 and o_j @ 2 respectively. Therefore, we should enforce a precedence constraint o_i @ 1 $\rightarrow$ o_j @ 2. Thus, an edge of 0-degree is introduced from o_i to o_j. In Fig. 4(c), o_j is assigned to DFG_1, but o_i is not scheduled into DFG_1 or DFG_2. The schedule is infeasible.

From the example, we know that, given $o_i \xrightarrow{d=1} o_j$, if o_j is scheduled at DFG_1, then o_i has to be scheduled into either DFG_1 or DFG_2. Also if o_i is scheduled into DFG_2, a constraint $o_i \rightarrow o_j$ of 0-degree has to be enforced between o_i and o_j in DFG_S. Our algorithm takes care of this issue by including a *pre-assignment* phase in the pipeline-schedule iteration. The pre-assignment phase assigns certain operations into DFG_{I+1} while we are dealing with the scheduling of pass I. The detailed algorithm will be described in Section 4.2.1.

Next, we will discuss the minimum achievable latency problem. Although both a trivial lower bound L_{min} and a trivial upper bound L_{max} on the achievable latency can be

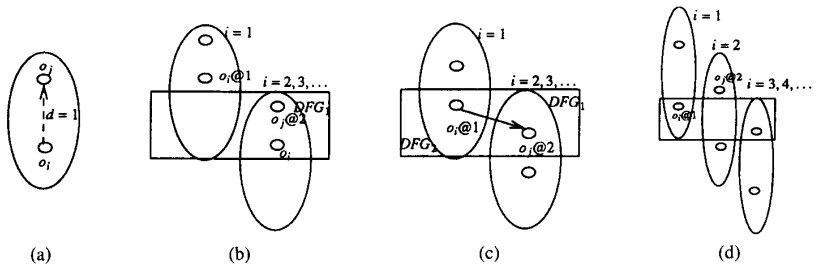

Fig. 4. An example with loop carried dependency. (a) *DFG* representation. (b) Both o_i and o_j are assigned to DFG_1. (c) o_j is assigned to DFG_1; while o_i is assigned to DFG_2. (d) An illegal schedule.

easily obtained, to determine the minimum latency is in general an NP-complete problem [11]. We explore the solution space by performing a *minimum-latency iteration* which repeatedly estimates a new latency and does a pipeline-schedule iteration until a feasible solution is found. The detailed strategy will be given in Section 4.2.2.

Fig. 5 shows the flow chart of our algorithm. A detailed description of the algorithm is given below.

4.2.1. The Pipeline-Schedule Iteration

The pipeline-schedule iteration is the kernel of our algorithm. It repeats itself I iterations by performing: *forward scheduling*, *pre-assignment*, *iterative folding* and *backward scheduling*. Forward scheduling and backward scheduling have been described in Section 4.1.

Pre-assignment: The pre-assignment phase is performed immediately after the forward scheduling of pass I. It assigns certain operations to DFG_{I+1}, and is called a *seed*, if it has a d-degree data precedence against an operation o_j scheduled in DFG_{I-d+1}. A pre-assigned operation produces data for its successor in DFG_S. It has to be executed before its consumers; otherwise, the producer-consumer precedence would be violated. Therefore, we enforce a constraint $o_i \rightarrow o_j$ of 0-degree on the o_i and o_j in DFG_S. After seeds are identified, all the unscheduled predecessors of the seeds are also assigned into DFG_{I+1}.

The pre-assignment phase plays an important role in our algorithm: (1) The pre-assigned operations and their consumers are very likely to be the elements of a dependency cycle or a strongly connected component. They have tighter precedence constraints and are in general more difficult to schedule. By pre-assignment, we can focus the attention on the critical parts. Thus, a feasible solution can be found quickly. (2) Once a dependency cycle or a strongly connected component is mapped to DFG_S, their precedences are projected to 0-degree. The problem is reduced to one without LCD's. (3) As more operations have been included in DFG_{I+1}, the backward scheduling can provide more informative bounds for the next forward scheduling.

Iterative folding: A partial schedule is said to be *convergent* if every operation is assigned to a step in the execution window. After some operations have been pre-assigned to DFG_{I+1}, the resulting partial graph may not be schedulable into the execution window (i.e., not convergent). During the iterative folding phase, we reorganize the DFG_S in order to find a convergent schedule. Three subtasks are performed:

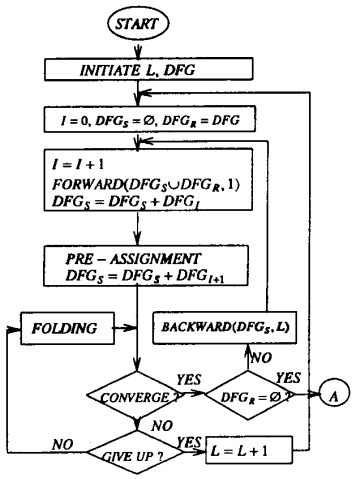

Fig. 5. The flowcharts of the algorithm.

(1) *Convergence Test:* If the result from applying a list scheduling DFG_S converges, we proceed to the next pass; otherwise, we have to decide whether to continue folding or to give up searching for a solution for the latency.

(2) *Folding Criteria:* Once we decide to perform a folding, those operations which have been scheduled beyond the execution window are folded. By means of folding, the operation o_i is moved from DFG_i to DFG_{i+1}. An arc is introduced from o_i to its consumer o_j to enforce the data precedence if o_i is in DFG_I, o_j is in DFG_J and there is an $(I - J)$ degree precedence between o_i and o_j in DFG. For example, when the o_i in DFG_1 of Fig. 4(b) is moved to DFG_2, we have to specify the data dependency $o_i \rightarrow o_j$ as that in Fig. 4(c) because $o_i \in DFG_2$, $o_j \in DFG_1$ and there is a 1-degree dependence between o_i and o_j.

(3) *Stopping Criterion:* Theoretically, if every operation of the DFG_S has been folded at least once, it would be impossible to find a solution anymore. However, it is unrealistic to do so because DFG_S

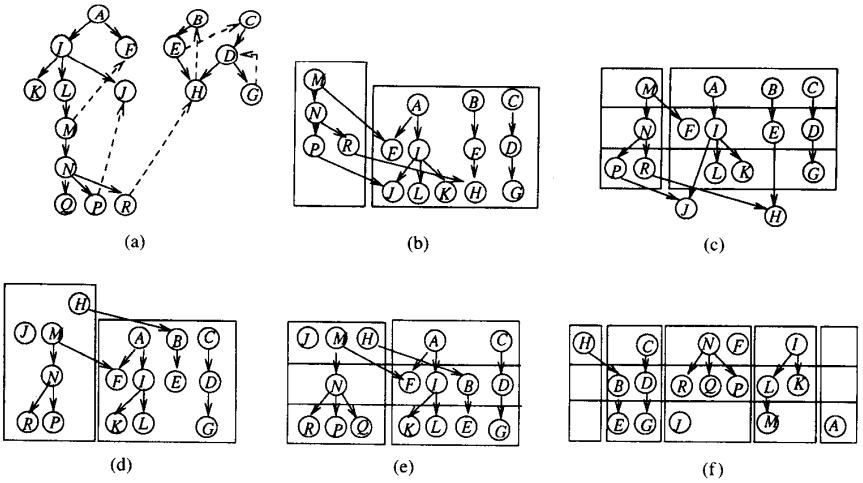

Fig. 6. Example to illustrate the proposed method. (a) A *DFG* to be scheduled. (b) The DFG_S after forward scheduling and the pre-assignment phase. (c) Results of the convergence test. (d) After folding is introduced. (e) The final result. (f) The result of OPT [8].

has a lot of operations to be detected and some operations are rarely folded, e.g., the predecessors of the operations in strongly connected components. Since the elements on a dependency cycle are more likely to be folded and the length of the longest cycle is not larger than $L_{\min}$, we terminate the folding iteration when the number of iterations exceeds $L_{\min}$.

4.2.2. *Example*

We will illustrate the algorithm using the example in Fig. 6(a). This example is first presented in [12] and is used by [8]. The *DFG* contains 17 operations. Its solid and dashed arcs denote data precedences of degree 0 and 1, respectively.

The lower bound on the achievable latency is 3 if there is no resource constraints, because there exists a loop (B @ 1 → E @ 1 → H @ 1 → B @ 2) whose execution delay is 3 and the total degree of edges is 1. The forward scheduling will schedule A, B, · · · , and L into DFG_1 (Fig. 6(b)). To meet the data precedence constraints, P and R (seeds) must be scheduled into DFG_2. Moreover, P @ 1 (R @ 1) must be scheduled one step earlier than J @ 2 (H @ 2). Accordingly, M and N must also be scheduled into DFG_2 because they are the predecessors of P and R, respectively. By combining DFG_1, the graph induced by $\{A, B, C, D, E, F, G, H, I, J, K, L\}$, and DFG_2, the graph induced by $\{M, N, P, Q\}$, with the arcs P @ 1 → J @ 2, R @ 1 → H @ 2 and M @ 1 → F @ 2, we get the partial graph, DFG_S, as shown in Fig. 6(b).

When we perform a list scheduling on DFG_S of Fig. 6(b), we found that the schedule cannot be converged within the execution window (Fig. 6(c)). To find a feasible solution, those operations (H and J) scheduled beyond the loop window shall be moved to the next block and a new arc (H → B) shall be introduced to specify the data dependencies (Fig. 6(d)). The iterative folding phase stops because the next convergence test succeeds. After a backward scheduling, the next forward scheduling schedules Q into DFG_2. Because every operation has been scheduled, the scheduling terminates with a latency of 3 and a delay of 6 (Fig. 6(e)). Fig. 6(f) shows the results by [8] in which code is spread over five subgraphs which has a much longer delay.

4.2.3. *The Minimum-Latency Iteration*

We perform the minimum-latency iteration to find the minimum achievable latency. It consists of three subtasks: (1) estimate the lower bound on the latency $L_{\min}$; (2) estimate the upper bound on the latency $L_{\max}$; and (3) repeatedly choose a latency and perform the pipeline-schedule iteration until a solution is found.

- *Estimate $L_{\min}$ and $L_{\max}$:* We set $L_{\min}$ as the value calculated using the method described in Section 3.2 and set $L_{\max}$ to be the number of control steps needed by performing a list scheduling on the unfolded *DFG*. For the example in Fig. 6, since there is a cycle B @ 1 → E @ 1 → H @ 1 → B @ 2 whose length is 3 and degree is 1, the minimum latency due to this cycle is 3. The upper bound of latency is 6 because the length of critical path of the non-pipelined loop is 6.
- *Find the minimum latency:* Obviously, the achievable latency falls within $[L_{\min}, L_{\max}]$. Although in the worst case we have to try every latency within $[L_{\min}, L_{\max}]$, the complexity is tempered by a binary search which requires $O(\log n)$ tries. However, due to the achievable latency is usually closer to $L_{\min}$ [11], we start with $L_{\min}$. If the pipeline-schedule iteration determines that a schedule with this latency is not achievable, the latency is increased by 1. This pro-

cess continues until a feasible solution is found. Experiments show that the linear search strategy takes only a few iterations in most of the cases.

4.2.4. *Complexity of the Algorithm*

The construction phase consists of three nested loops: (1) folding iteration, (2) pipeline-schedule iteration and (3) minimum-latency iteration. The folding iteration repeats at most L times where each iteration employs a list scheduling. Therefore, its complexity is $O(L \times n \log n)$. The complexity of the pre-assignment and the basic scheduling are $O(n)$ and $O(n \log n)$, respectively. Suppose the schedule finished after I passes, the overall complexity of the pipeline-schedule iteration is $O(I \times L \times n \log n)$. Consequently, the complexity of the minimum-latency iteration is $O(I \times L \times n \log^2 n)$ in the worst case and is $O(I \times L \times n \log n)$ in average.

V. Experimental Results

The proposed method has been implemented in a C program running on a SUN-SPARC station. It is called PipeLining Scheduler (PLS). The solutions produced by an integer programming approach [13] are used as a comparison purpose. The data path is constructed by STAR [14].

We compare the results of PLS with those by other systems on the following benchmarks: (1) Cytron's example [12], (2) the fifth-order digital wave filter [15] and (3) the fast discrete cosine transform [16]. We assume that multipliers take two cycles while adders and subtractors take one cycle each to complete. In order to have a fair comparison, we do not restructure the data flow graph of the test cases. The solutions by ALPS are also included. A "○" in the tables means that the result is the same as the optimum.

5.1. *The Cytron's Example*

The data flow graph of the example is shown in Fig. 6(a). This example was first presented in [12]. We have used this example throughout the paper to explain the concept of our pipeline scheduling algorithm. The example has 17 operations which are of the same type and take one control step to execute. The length of the longest path is 6 and the longest dependence cycle is 3. Table I compares our results with those by the percolation based scheduler (PBS) [7] and the ATOMICS [10]. Only PLS obtains minimum delay for all the cases. PBS begins with a maximum parallelism pattern (shown in Fig. 6(f)) and then applies a heuristic to adjust pattern to fit the given resource constraint. Although the minimum latency is achieved, the delay increases because the operations are scattered across five iterations. The results of ATOMICS are obtained based on the loop folding algorithm presented in [10].

TABLE I
Cytron's Example

Operators	Latency	Delay			
		Optimum	PLS	ATOMICS	PBS
unlimit	3	6	○	○	11
6	3	6	○	7	11
5	4	6	○	8	14
4	5	6	○	9	17
3	6	6	○	○	20

TABLE II
Fifth Order Digital Wave Filter with Non-Pipelined Multiplier

System	Spaid		PLS					
Adders	3	2	3	2	2	2	1	1
Multipliers	2	1	2	2	1	1	1	1
Buses	6	6	6	6	6	4	4	2
Latency	17	21	16	17	19	19	28	35
Delay	—	—	17	16	20	20	15	15
Completion	—	—	18	19	21	21	28	35

TABLE III
Fifth Order Digital Wave Filter with Pipelined Multiplier

System	Spaid							PLS				
Adders	4	3	2	2	2	1	2	3	2	2	1	1
Multipliers	2	2	1	1	1	1	1	1	1	1	1	1
Buses	7	6	6	5	4	4	2	6	6	4	4	2
Latency	16	17	18	19	21	29	39	16	17	19	28	34
Delay	—	—	—	—	—	—	—	17	16	15	15	15
Completion	—	—	—	—	—	—	—	18	19	21	28	34

5.2. *The Fifth Order Digital Wave Filter*

This example has a relatively large number of loop carried data dependencies as well as intra-loop dependencies. The critical path length of the 0-degree subgraph is 17 and the length of longest cycle is 16. The strong data dependency has limited the throughput even given unlimited resources. Tables II and III compare the results of Spaid [9] and PLS by using a non-pipelined and a pipelined multiplier, respectively. The experiments assume a two-phases timing bused architecture [9], [14], in which read and write are performed at different phase. Given the same resource constraint, our approach (i.e., perform retiming while scheduling) obtains better results. We also report the completion time and delay time of our schedules. The *completion* (delay) time is the duration from the first input node to the last sink (output) node of an iteration.

5.3. *Fast Discrete Cosine Transform (FDCT)*

This example is found in [16]. It contains 42 operations: 16 multiplications, 13 additions and 13 subtractions. The length of the critical path is 8. Therefore, the maximal degree of parallelism is 5.3. Due to the nature

TABLE IV
THE PIPELINED SYNTHESIS OF THE FAST DISCRETE COSINE TRANSFORM WITH PIPELINED MULTIPLIERS

Busses		58	31	20	16	12	11	11	8	7	6	7	6	4	2
Adders		13	7	5	4	3	3	2	2	2	2	1	1	1	1
Subtractors		13	7	5	4	3	3	2	2	2	2	1	1	1	1
Multipliers		16	8	6	4	4	3	3	2	2	2	2	2	1	1
PLS	Latency	1	2	3	4	5	6	7	8	9	11	13	14	16	32
	Delay	8	9	11	12	11	12	12	14	13	14	15	14	28	32
	CPU(sec.)	0.1	0.13	0.18	0.1	0.15	0.13	0.18	0.1	0.22	0.19	0.17	0.09	0.21	0.1
Sehwa	Latency	1	2	4	4	5	7	7	9	12	13	13	15	22	32
	Delay	8	10	11	12	12	11	14	18	21	22	26	15	34	32

of large mobility of the operations, the solution space is very large. It prevents a solution from being found optimally by the ILP method. Table IV compares the results by PLS and Sehwa.

In this example, buses are also specified as part of resource constraints. We assume that a bus can broadcast a data to many function units at a time step. Sehwa fails to find a minimum latency in about half of the cases. For those cases which Sehwa finds minimum latency, a longer delay is used. The reason that Sehwa performs poor is that it may find the functional unit for an operation at one step; while buses are available at another step. Our approach can overcome such issue because operations can be flexibly moved.

Mallon *et al.* [16] use the case with 2 adders, 2 multipliers, and 2 subtractors to illustrate their algorithm. Starting from an initial solution (by an algorithm called ASAP), they apply an optimization strategy that tries to compress the initial solution by swapping folded operations with others of the same type that have a lower folding index. The latency and delay of the initial solution are 8 and 14, respectively. The delay of the final solution is 12. However, the result is based on the assumption that a multiplier and an adder (or a subtractor) can be chained within a cycle. PLS obtains a result with delay 11 under the assumption that the multiplier and adder each takes one cycle to complete.

VI. CONCLUSION

In this paper, we have presented an algorithm for pipelining the execution of a loop in the presence of loop carried dependencies. Our method optimizes both the initiation interval and the turn around time of the pipeline. Given the constraint on the number of functional units and buses, we first determine an initiation interval and then incrementally partition the operations into several blocks. Experiments on benchmark examples show that the new approach gains a considerable improvement over those by previous approaches.

REFERENCES

[1] M. C. McFarland, A. C. Parker, and R. Camposano, "The high-level synthesis of digital system," in *Proc. IEEE*, vol. 78, pp. 301–318, Feb. 1990.
[2] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Margan Kaufmann, 1990, pp. 327.
[3] N. Park and A. C. Parker, "Sehwa: A software package for synthesis of pipelines from behavioral specifications," *IEEE Trans. Computer-Aided Design*, vol. 7, pp. 356–370, Mar. 1988.
[4] 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.
[5] Ki Soo Hwang, Albert E. Casavant, Ching-Tand Chang, and Manuel A. d'Abreu, "Scheduling and hardware sharing in pipelined data paths," *Proc. ICCAD-89*, pp. 24–27, Nov. 1989.
[6] E. M. Girczyc, "Loop winding—A data flow approach to functional pipelining," *Proc. IEEE ISCAS*, pp. 382–385, May 1987.
[7] R. Potasman, J. Lis, A. Aiken, and A. Nicolau, "Percolation based synthesis," in *Proc. 27th Design Automation Conf.*, pp. 444–449, 1990.
[8] A. Aiken and A. Nicolau, "Optimal loop parallelization," in *Proc. 1988 ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, 1988.
[9] B. S. Haroun and M. I. Elmasry, "Architectural synthesis for DSP silicon compiler," *IEEE Trans. Computer-Aided Design*, vol. 8, pp. 431–447, Apr. 1989.
[10] G. Goossens, J. Rabaey, J. Vandewalle, and H. De Man, "An efficient micro-code compiler for applications specific DSP processors," *IEEE Trans. Computer-Aided Design*, vol. 9, pp. 925–937, June 1990.
[11] M. S. Lam, "A systolic array optimizing compiler," Ph.D. dissertations, Carnegie Mellon Univ., 1989.
[12] Cytron, R., "Compiler-time scheduling optimization for asynchronous machines," Ph.D. dissertation, Univ. Illinois at Urbana-Champaign, 1984.
[13] C. T. Hwang, Y. C. Hsu, and Y. L. Lin "Optimum and heuristic data path scheduling under resource constraints," in *Proc. 27th Design Automation Conf.*, pp. 65–70, July 1990.
[14] F. S. Tsai and Y. C. Hsu, "DataPath construction and refinement," in *Proc. ICCAD-90*, pp. 308–311, 1990.
[15] S. Y. Kung, H. J. Whitehouse, and T. Kailath, *VLSI and Modern Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, pp. 258–264, 1985.
[16] D. J. Mallon and P. B. Denyer, "A new approach to pipeline optimisation," in *Proc. European Design Automation Conf.*, Mar. 1990.

Cheng-Tsung Hwang received the B.S. degree in Computer Science from Tung-Hai University, Taichung, Taiwan, in 1987, and the Ph.D. degree in computer science from Tsing Hua University, Taiwan, in 1992.

He has been doing postgraduate research in the Computer Science Department of the University of California, Riverside during 1991–1992. His research interests include silicon compilation and optimization in VLSI design.

Yu-Chin Hsu, for a photograph and a biography please see page 424 of the March 1993 issue of this TRANSACTIONS.

Youn-Long Lin, for a photograph and a biography please see page 424 of the March 1993 issue of this TRANSACTIONS.