

## A Fast Implementation for Recurrent DSP Scheduling Using Final Matrix

DANIEL Y. CHAO

*Department of Management Information Systems  
National Cheng Chi University  
Taipei, Taiwan 116, R.O.C.  
E-mail: yaw@mis.nccu.edu.tw*

The scheduling theory of Heemstra de Groot et al. is supplemented by extending the Final Matrix's usefulness beyond finding iteration bounds, critical loops and subcritical loops of recursive data flow graphs (RDFGs) to scheduling. DFG is a special case of Petri nets (PN). Hence we apply the cycle time theory of PN to the scheduling of DFG. Contributions include: (1) development of explicit formulas for slack time, scheduling ranges and update, and static rate-optimal time scheduling based on entries of the final matrix; (2) development of a fastest processor assignment algorithm based on the rate-optimal static scheduling without unfolding while considering abnormal cases in which iteration bounds are fractional or smaller than some node execution times; (3) discovery of a new anomaly in addition to the above two cases; (4) development of a user-friendly Final-matrix based Integration Tool (*FIT*) to view critical and subcritical loops, iteration bounds, scheduling ranges, and level and processor assignment diagrams based on a single tool "final matrix" rather than other tools; (5) elimination of redundant steps such as the construction of inequality graphs; (6) development of a proof showing that the *ALAP* and *ASAP* fixed-time schedulings satisfy the firing rule; and (7) thousand-fold faster (linear time complexity) processing compared to others and use of fewer processors in the case of large DFGs.

**Keywords:** scheduling, concurrent processing, data flow graph (DFG), final matrix, iteration bound, loop bound, critical loop, CAD

### 1. INTRODUCTION

In a multiprocessor environment, the recursive Data Flow Graph (RDFG) [1] is frequently used to model specifications of real time applications performing repetitive tasks, such as image processing, and to express the available concurrence. DFGs are directed graphs, where directed edges model the precedence constraints between nodes. Each node accepts one input data from each of its input node, executes some tasks and outputs one data to each of its output nodes. External data to the system are sampled periodically; the corresponding period is termed the  **$d$  = iteration period** [2] as it is also the time period of a DFG between two successive executions of a node.

System throughput is improved by reducing the sampling period as more processors are used to increase concurrency. It is well-known that this improvement cannot be continued indefinitely, and that there is a lower bound to the iteration period. Such an iteration

---

Received November 7, 1997; revised June 15, 1998; accepted August 11, 1998.  
Communicated by Shing-Tsaan Huang.

period is the shortest one and is termed the **iteration bound**. The corresponding scheduling is called *rate-optimal* [2]. A RDFG contains a number of loops, some of which, called **critical loops**, determine the iteration bound. This is similar to the case of nonrecursive DFGs, whose performance is determined by the critical path, scheduling techniques for which have been well-developed. Much less overhead is required to schedule nodes at compile or design time, i.e., static scheduling, compared with run-time scheduling. An active research goal has been to achieve the maximum throughput with the minimum number of processors using appropriate scheduling. There are two aspects of scheduling: time and processors. Each node must be assigned a starting time and a processor to execute it; optimization is an NP-complete problem [1]. Heemstra de Groot et al. have discussed various scheduling methods and invented an effective scheduling method [1, 3] to achieve the minimum iteration period given a set of processors (called *maximum-throughput scheduling with limited resources*), and to find the minimum number of processors for a given iteration period (called *fixed rate scheduling*). This work is concerned only with fixed rate scheduling, which is briefly reviewed to show the advantages of the method proposed by Heemstra de Groot et al. The single-iteration method [4, 5] exploits concurrence within a single iteration. The direct-blocking method [6, 7] exploits concurrence across a number of iterations. These two methods do not fully exploit all the available concurrence; therefore, the number of processors is not optimized. The fixed-rate method assigns processors to nodes given the iteration period. An example is Parh's approach [2], which first translates the DFG into an equivalent perfect-rate RDFG, and then optimally unfolds it. The disadvantage is the extra memory required. There are two other such approaches: (1) The maximum spanning tree was proposed by Renfors and Neuvo [8-10]. This method does not explicitly try to optimize the number of processors. (2) Search of cyclo-static schedules [11] fixes both the iteration period and the number of processors. The displacement in the space-time coordinates of successive iterations is recorded in a principle lattice vector with no efficient way to find the optimal one. This method does not guarantee a feasible schedule, and the worst case computational complexity is exponential.

The main idea of Heemstra de Groot et al.'s work lies in the concept of the scheduling range. Nodes on a critical loop have fixed execution times within each iteration. Nodes on noncritical loops, however, have flexibility in terms of their execution times within an iteration; that is, there exist time range ranges for the scheduling of such nodes. Appropriate scheduling of such nodes, subject to the constraint of the scheduling range, may lead to an optimized number of processors and a minimum iteration period. They calculate the iteration bound and the mobility (the length of the scheduling range) of each node separately. The iteration bound is computed [12-14] based on the minimum cost-to-time ratio cycle algorithm described in [15]. They first pack the nodes into a few levels as possible (i.e., construct the level diagram) to determine the start execution time of each node during each iteration. Then they decide which processor to assign for each node using a heuristic. The inequality graph is constructed to find the scheduling range without explicit formulas. The best candidate for scheduling next in their scheduling **heuristic** holds the shortest scheduling range and can be searched in a linear fashion. Once the node's execution time is fixed, the scheduling range of nodes (not yet scheduled) is updated using the inequality graph, and this process is continued until all the node's execution times are fixed.

This work is based on the work of Heemstra de Groot et al. [7]. They employ a level diagram to assign operations to processors. The X-axis of the diagram is divided into  $I$  equivalence classes, where  $I$  is the iteration period. Operations are assigned to processors.

To search the time slots for an operation in the levels is optimized. Finally, operations are assigned to processors. To search the time slots for an operation in the level diagram, the scheduling-range chart is used, which contains information on the range within which the operation can be scheduled. An inequality graph is used to find and update scheduling-range.

They, however, do not consider scheduling **abnormal** cases of (1) fractional iteration bounds (IB) and (2) large node execution times (greater than IB). The consequence is that the scheduling of Heemstra de Groot et al. achieves less processor utilization than do those of Wang et al. [16] and Jeng et al. [17], who minimize the unfolding factor to include the above anomalies.

Wang et al. [16] considered case (2) while minimizing the number of unfolding fact  $\beta_1$  such that  $\beta_1 I > x_{\max}$  and  $(\beta_1 - 1)I < x_{\max}$ , where  $x_{\max}$  is the maximum node execution times among all the nodes. The idea is that all fully static periodic schedules must be upper bounded by a *cutofftime*. Comparing all such schedules enables one to minimize the number of processor and leads to faster scheduling.

Jeng et al. [17] further considered case (1) by multiplying the above  $\beta_1$  by another factor  $\beta_2$  such that  $\beta = \beta_1 \beta_2$  to satisfy the conditions that  $\beta I > x_{\max}$  and  $\beta I \in N$  is an integer. These two approaches still incur hardware and software overheads associated with unfolding.

We eliminate unfolding while considering both anomalies. Further, to schedule, one must know the iteration bound. All other approaches assume a known iteration bound and do not exploit the information available during the process of finding the iteration bound. As a result, their schedulings take too much time compared with ours, where scheduling is based on the Final Matrix [14] obtained after finding the iteration bound. Further, we treat a third anomaly and avoid the construction of inequality graph and most updates.

The scheduling method of Heemstra de Groot et al. does not consider the initial scheduling; it considers only steady-state scheduling. As a result, idle or transient periods may appear in the initial portion of the schedule. We proposed a theory in this paper to eliminate such initial transient periods.

To make this paper self-contained, we review the theory of the final matrix [12, 14] in section 2. Section 3 briefly explains our approach. An example of multiprocessor scheduling based on critical and subcritical loops is given in section 4. Section 5 derives the slack times of subcritical loops and scheduling ranges plus their node updates. Section 6 presents the formulas for fixed-time and initiation time scheduling without the transient period, followed by processor scheduling in section 7. The advantages of our CAD tool *FIT* and benchmark results are given in section 8. Finally we conclude this paper in section 9.

## 2. PRELIMINARIES

### 2.1 Iteration Bound and Critical Loop

To simplify the presentation, we assume that all the nodes of a DFG are on certain loops. As shown in Fig. 1, an edge  $E_i$  carrying data to a node bears a symbol “ $x_i D$ ” ( $x_i$  delay or register elements) or “ $D$ ” where “ $x_i$ ” is a positive integer indicating the difference of iteration numbers between the input data into the edge and the output data from the edge – an inter-iteration precedence relationship between a data-accepting node and a data-outputting node. If there is no delay element on an edge from node  $n_i$  to node  $n_j$ , then the  $n$ th execution of tasks in  $n_j$  involves data from the  $n$ th execution of  $n_i$  – an intra-iteration precedence relationship between  $n_i$  and  $n_j$ .

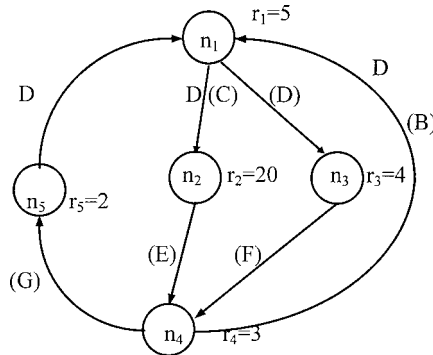


Fig. 1. A DFG with five nodes and seven arcs.

We follow [12] in defining the following terms. Let  $T_k = \sum_{n_i \in L_k} r_i$  denote the sum of the execution times  $r_i$  of nodes  $n_i$  in loop  $L_k$ .  $N_k = \sum_{E_i \in L_k} x_i$  denotes the total number of delay elements  $x_i$  of every edge  $E_i$  in loop  $L_k$ . **Loop Bound**  $LB_k$  of loop  $L_k = \left\{ \frac{T_k}{N_k} \right\}$  is the total loop computation time,  $T_k$ , divided by the number of delay elements,  $N_k$ , inside loop  $L_k$ . **Iteration Bound** ( $I = \max \left\{ \frac{T_k}{N_k}; k = 1, 2, \dots, q \right\}$ ,  $q$  being the number of loops) of a system is the lower bound on the achievable iteration period. The loop with the largest loop bound is called the **critical loop**. There are four loops in Fig. 1, and loop  $n_1 - n_2 - n_4 - n_5 - n_1$  is the **critical loop** with the maximum loop bound  $I = \frac{30}{2} = 15$ . The **Slack Time** of a loop  $L_k$  with the iteration period  $= R$  is  $S_k = RN_k - T_k$ . The **mobility** of a node is the magnitude of its scheduling range.

Let  $n$  be the number of nodes and  $R$  be the guessed IB. Construct an  $n \times n$  matrix (Fig. 2), called the **Initial Matrix**, so that each entry  $q_{ij} = Rx_k - r_i$ , if  $n_i$  connects to  $n_j$  through  $E_k$  and  $\infty$  if they are not directly connected. Based on the initial matrix, a series of intermediate and the final matrix  $Q^{(m)}$ ,  $m = 2, 3, \dots, f$  can be found according to Floyd-Warshall's Algorithm [15] as shown in Fig. 3. When  $m = f$ ,  $Q^{(f)}$  is the final matrix obtained, and its entry  $q_{ij}^{(f)}$  denotes the shortest distance between  $n_i$  and  $n_j$ . There are three cases associated with the diagonal entries as follows:

**Theorem 1** [18]: In the final matrix  $Q^{(f)}$ , if the diagonal entries are (1) all positive,  $R > I$ ; (2) some negative,  $R < I$ ; and (3) one or more zero and the rest positive,  $R = I$ .

$$Q^{(1)} = \begin{bmatrix} \infty & 10 & -5 & \infty & \infty \\ \infty & \infty & \infty & -20 & \infty \\ \infty & \infty & \infty & -4 & \infty \\ 12 & \infty & \infty & \infty & -3 \\ 13 & \infty & \infty & \infty & \infty \end{bmatrix}$$

Fig. 2. The initial distance matrix.

$$Q^{(1)} = \begin{bmatrix} \infty & 10 & -5 & -10 & \infty \\ -8 & \infty & \infty & -20 & -23 \\ 8 & \infty & \infty & -4 & -7 \\ 10 & 22 & 7 & \infty & -3 \\ 13 & 23 & 8 & \infty & \infty \end{bmatrix} \quad (\text{I})$$

$$Q^{(2)} = \begin{bmatrix} 0 & 10 & -5 & -10 & -13 \\ -10 & 0 & -15 & -20 & -23 \\ 6 & 18 & 3 & -4 & -7 \\ 10 & 20 & 5 & 0 & -3 \\ 13 & 23 & 8 & 3 & \infty \end{bmatrix} \quad (\text{II})$$

$$Q^{c(2)} = \begin{bmatrix} 0 & 10 & -5 & -10 & -13 \\ -10 & 0 & -15 & -20 & -23 \\ 6 & 16 & 1 & -4 & -7 \\ 10 & 20 & 5 & 0 & -3 \\ 13 & 23 & 8 & 3 & 0 \end{bmatrix} \quad (\text{III})$$

Fig. 3. The computed matrices.

Using binary search, we can obtain  $I$  within tolerable error. Nodes on critical loops can be identified, and the corresponding loop bound constitutes the  $I$ . Then we can construct the final matrix with  $R = I$ . Refer to [14] for more details.

## 2.2 Subcritical Loop

Every node has a shortest loop through itself. Such a loop is called the **subcritical loop** through the node if the loop is not a critical one. For the RDFG in Fig. 1, all nodes except  $n_3$  are on the subcritical loop:  $n_3 - n_4 - n_5 - n_1 - n_3$ . To form a large continuous segment, we schedule nodes on a subcritical loop with less slack time before any node on another subcritical loop with larger slack time (also used in the Scheduling algorithm by Parhi). If two subcritical loops tie with each other, all the nodes of one of these two subcritical loops are scheduled before any node on the other subcritical loop.

As shown in [14], identifying subcritical loops is crucial in applying the look-ahead technique to reduce  $I$ . Hence, a procedure for finding subcritical loops has also been developed [14]. It is useful because our scheduling algorithms schedule subcritical loops in increasing order of slackness.

## 3. THE APPROACH

Using the final matrix obtained in finding the iteration bound, we develop formulas to find the slackness, the scheduling ranges and their updates, and **Fully Static As Soon As Possible (FSASAP)** and **As Late As Possible (FSALAP)** scheduling times for all nodes. We prove that they satisfy the firing rules. These two schedulings conform to the principle of forming large continuous operation regions as most nodes on a subcritical loop will fire one after the other, thus reducing the number of fragmented holes and forming a larger

fragmented hole. This provides us with a good initial trial to construct the level diagram, where each level ranges from 0 to  $I$ , the iteration bound. The result is near optimal. Next, we apply a simple algorithm to reduce the number of processors. This differs from [1], where the time schedules of all nodes are continuously updated and cannot be determined until the level diagram is completely constructed. As a result, our approach runs faster than others which require repeated updating of schedules.

[1] employed inequality graphs to find scheduling ranges and their updates. We found that this inequality graph is redundant to be replaced by the formula based on the final matrix. Thus, we can eliminate this extra step of constructing inequality graph, which has a time complexity of  $O(ne)$ , where  $n$  and  $e$  are the number of nodes and edges, respectively. Each time a node's time scheduling is fixed, other unscheduled nodes' scheduling ranges must be updated. Again, we can eliminate the construction of inequality graphs to update scheduling ranges based on the final matrix. Further, we can find some specific periodic rate-optimal time scheduling based on the final matrix to avoid initial time transients; that is, in the first iteration, some nodes may not be executed.

Thus, unlike all other approaches which separate searching of critical loops and iteration bounds from the scheduling, we integrate them by applying the *final matrix* to search critical loops, iteration bounds, and time and processor scheduling. This results in great speed-up of processor scheduling; hence an ideal CAD tool for processor assignment should be final-matrix based.

We observe that in most cases, the fraction of nodes that is on critical loops is over one-half of all the nodes in the DFG. For all nodes on the same critical loop, we need not fix their scheduling times or the associated scheduling updates. This simplifies processor scheduling as we can simply place successive nodes on the critical loop one after another.

Afterwards, we deal with subcritical loops and nodes not in any loops; the start execution times of the latter can be any times since its scheduling range is infinitely large, and the scheduling is in decreasing order of node execution times. This leaves many fewer nodes to deal with in the level diagram. Minimization of the number of processors can be more readily achieved.

Even though we start with a good initial trial, the number of levels may not be a minima. We reduce it in two ways. First, after scheduling subcritical loops, we may eliminate the last few levels since their nodes have relatively large slackness. We reschedule only segments purely in the subcritical loop scheduled last but not in other loops scheduled earlier; the two end nodes of the segment have been scheduled earlier. Rescheduling is performed under the condition that these two nodes' scheduling is not perturbed. This minimizes the scheduling updates when we schedule the first node in the segment; the remaining nodes in the segment can be continuously scheduled without update calculation. Second, we schedule type "n" nodes in an intelligent fashion to minimize the number of levels.

To minimize the number of processors, the utilization of a processor must be as large as possible. This can be achieved by reducing the number of fragmented holes. In other words, the time period during which it continuously operates must be as long as possible. This is the primary principle that we adopt in our algorithm, resulting in fewer processors required than in [16] and performing better than the algorithms in [1, 19].

If the nodes of a critical loop are assigned to a processor, then it operates continuously without any idle period and achieves 100% utilization. Thus, the algorithm schedules critical loops first. Since nodes on a critical loop are executed one after another, it is easy and fast to schedule them in a continuous fashion without the need to search empty holes. For

the same reason, we then schedule subcritical loops based on their slackness. Finally, we schedule type “n” nodes defined as not being on any loops. Since the mobility of these nodes is infinite, they can be scheduled at any moment within the iteration period. Thus, we can reduce the number of fragmented holes by filling them with type “n” nodes. Following the principle of forming a large continuous operation region, (1) we schedule type “n” nodes in order of decreasing node computation time; (2) when inserting the node into a fragmented hole, we always fill the hole from the its left end; and (3) if a fragment hole is too small to fit the type “n” node under the constraint that the level must contain at least one node which is not type “n”. This works well for large DFGs with many “n” nodes and is the sole reason for using fewer processors compared to the case in [16].

We propose a simpler algorithm without unfolding while taking scheduling anomalies into consideration. This is based on the observation that during one iteration of the system, each processor may complete several iterations. If we just show the node assignment during just one iteration in each processor, the amount of unfolding can be reduced, and so can the memory and displace space. Based on Chen’s technique, unfolding is still required because the iteration period of a processor may be more than one  $I$ .

Unfolding was not required in [1] because they deal with only one iteration bound. This is not incorrect if all node’s execution times are smaller than  $I$ . If a node cannot be fitted into a processor, we just assign it to a new processor. This, however, may increase the number of processors required. For instance, consider a critical loop. We can assign as many processors as there are tokens or delay elements in the loop, and each processor is fully utilized. Using [1], the iteration period of each processor equals  $I$ . This is impossible if some node’s execution times are greater than  $I$ , or if  $I$  is fractional. As a result, more processors are required. There is a new anomaly not reported in [1]. It arises from the fact that even though  $I$  is an integer, one iteration bound may not be completely, filled by nodes. Either the last node ends beyond  $I$  or some holes exist in the period of one  $I$ . Note that in all three cases, the anomalies can be detected by noting that the end of some node execution time reaches beyond  $I$  while some node executes at  $t = 0$  on the same processor. As a result, unlike Chen’s approach, a uniform approach is proposed to deal with all anomalies, which are treated in the processor assignment phase following completion of the level diagram. This approach eliminates unfolding while considering all anomalies by **allowing a node to be assigned to multiple processors** while still maintaining *static rate-optimal* scheduling.

This new technique has been incorporated in our CAD tool *FIT* for processor scheduling of DFGs. In the CAD tool, the designer will be able to view the scheduling ranges and processor assignments graphically by clicking on some menu buttons.

#### 4. EXAMPLES OF SCHEDULING

To give the reader an idea of scheduling and better insight into the theory presented later, we present an example of scheduling. After locating the critical loop, we may develop a schedule to achieve the maximum throughput or minimum iteration period. One scheduling method is **ASAP (As Soon As Possible) scheduling**, where any node is executed as soon as the node is executable. An example of *ASAP* scheduling based on the critical loop found and the calculated iteration bound, for the RDFG in Fig. 1, is shown in Fig. 4. We allocate two processors, **P1** and **P2**, for nodes on the critical loop since there are two independent data samples (i.e.,  $N = 2$ ) to execute the nodes in the critical loop simultaneously.

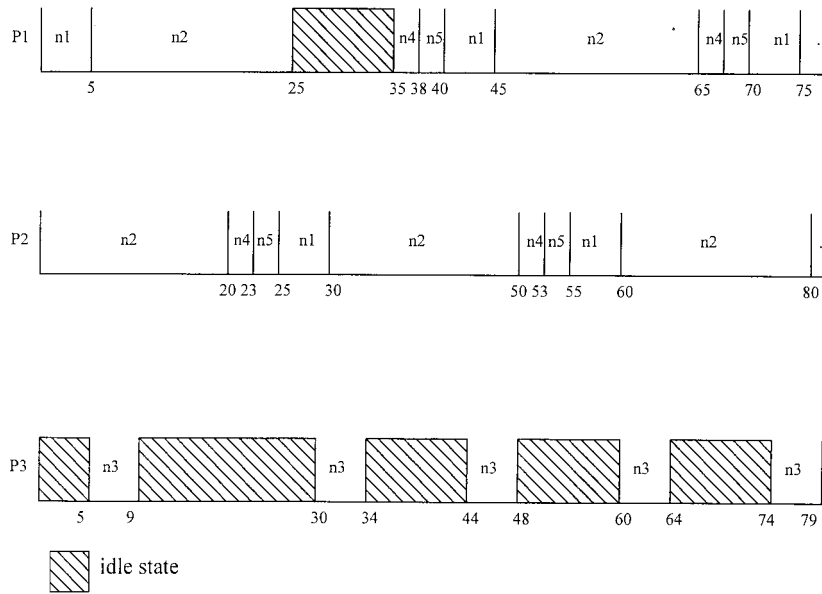


Fig. 4. The ASAP scheduling of the DFG in Fig. 1.

We assign another processor (**P3**) to  $n_3$ . After  $t = 35$ , the throughput of the system is maintained at one iteration per 15 units of time; i.e., the iteration bound is 15. Note the processor **P1** is idle only during the time period from time 25 to 35, that **P2** has no idle period and that **P3** is idle most of the time. After  $t = 35$ , both **P1** and **P2** have a throughput of one iteration per 30 time units, and the system is said to reach **steady state**. Both processors have an iteration period of  $2I = 30$ .

Thus, in general, nodes on the critical loop are executed at a fixed time within each iteration period on a processor after the initial transient period. Since  $I$  equals the loop bound of the critical loop  $L_c$ , i.e.,  $I = \frac{T_c}{N_c}$ ,  $N_c$  samples of data must be processed during  $T_c$  period of time, and we assign one processor for each data sample (or delay) on  $L_c$ . Other nodes which are not on the critical loop should be executed in such a fashion so that the fixed timings of the nodes on the critical loop do not change. Hence the general principle in scheduling is that one assigns as many processors as there are delay elements on the critical loop. Processors are assigned to other nodes on subcritical loops so as not to disturb the schedule of nodes on the critical loop. Subcritical loops with shorter slack times (hence, less mobility) have higher priority for scheduling. Processors assigned to subcritical loops may be idle during one iteration due to the fact that their loop bounds are smaller than the maximum loop bound. Because of this idling, the execution of a node on a subcritical loop can be more flexible, and there exists a lower bound and an upper bound (i.e., scheduling range) of its execution time during each iteration. The larger the deviation of its loop bound from the maximum, the more flexible the scheduling for nodes on the subcritical (but not on the critical) loop. We can use the final matrix ( $Q^{(f)}$ ) to determine the scheduling range for any node  $n_a$  as described in the next section.

Although the above scheduling achieves the maximum throughput or minimum iteration period, it is not fully static; i.e., the scheduling is determined at run time rather than compile time, incurs much overhead. Also there is an idle period (hole) in processor **P1** and **P1** executes nodes only on the critical loop. Such a hole can be eliminated with **ALAP** (As Late As Possible) scheduling (Fig. 5), where the execution of an initially executable node is delayed as much as possible without making the iteration period longer than the iteration bound. The delay is such that the data samples on each critical loop are evenly distributed, and the time interval between any two successive data samples is  $I$ . In Fig. 5, the execution starting times of these processors are staggered, and processor **P2** starts execution first. Each such processor will execute nodes on the critical loop without idling periods. In general, it is not obvious which initially executable nodes should delay its execution and which should not. This will be dealt with in Theorem 4.

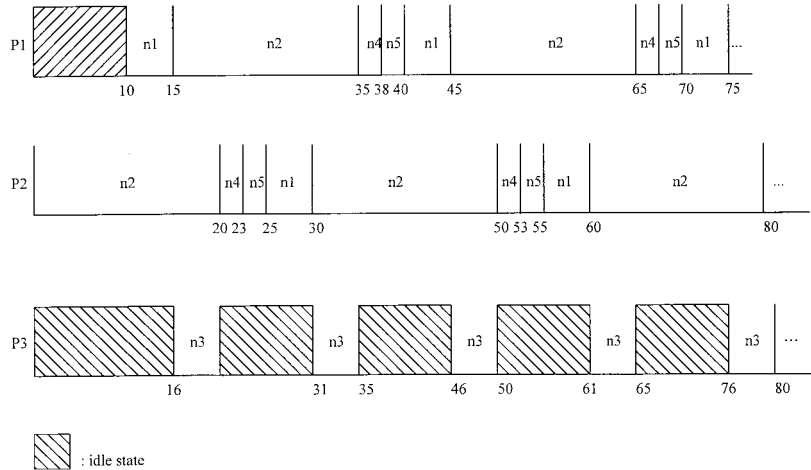


Fig. 5. The ALAP scheduling of the DFG in Fig. 1.

### 5. SLACKNESS, SCHEDULING RANGE AND ITS UPDATE

In the rest of this paper, let  $n_r$  denote the reference node which executes at the beginning of each iteration under steady state. The execution times of all other nodes during each iteration can be expressed relative to  $n_r$ .

#### 5.1 Slack Time for $R = I$

**Lemma 1:** When the guessed iteration period equals the iteration bound, i.e.,  $R = I$ , (a) the slack time of the subcritical loop through a node not on any critical loops equals the diagonal entry of the node; (b) the slack time of a critical loop is zero; (c) the slack time of any loop through a node is no less than the diagonal entry of the node.

**Proof:** (a) The slack time of a loop, by definition, represents the distance from a node  $n_k$  of the loop to the node itself on the loop. Now the diagonal entry ( $a_{kk}$ ) of  $n_k$  in the final matrix corresponds to the shortest distance and hence the slack time of the subcritical loop from the node to itself; (b) comes from the fact that the diagonal entries (slack times) for nodes on the critical loop are zeros; (c) comes from the fact that the distance from a node to itself on any nonsubcritical loop through the node is larger than that of the subcritical loop.  $\square$

## 5.2 Petri Net (PN) Model and Inequality Graph

Our theory concerning the scheduling range is based on the inequality graph of Heemstra de Groot et al., which is based on the forward and backward relations. Heemstra de Groot et al.'s token flow concept behind the above two relations can be better understood by recognizing that RDFG is a special case of PN as discussed below and followed by the inequality graph.

The program for Fig. 1 is as follows:

Program 1:

```

initial conditions:  $n_5n_1(0)$ ,  $n_1n_2(0)$ , and  $n_4n_1(0)$ ;
for each  $\{k = 1, \text{ to } \infty\}$  {
 $n_1n_2(k) = f_{n_1n_2} [n_4n_1(k-1), n_5n_1(k-1)]$ 
 $n_1n_3(k) = f_{n_1n_3} [n_4n_1(k-1), n_5n_1(k-1)]$ 
 $n_2n_4(k) = f_{n_2n_4} [n_1n_2(k-1)]$ 
 $n_3n_4(k) = f_{n_3n_4} [n_1n_3(k)]$ 
 $n_4n_1(k) = f_{n_4n_1} [n_2n_4(k), n_3n_4(k)]$ 
 $n_4n_5(k) = f_{n_4n_5} [n_2n_4(k), n_3n_4(k)]$ 
 $n_5n_1(k) = f_{n_5n_1} [n_4n_5(k)]$ 
},

```

where  $xy(n)$  ( $x, y = n_1, n_2, n_3, n_4$  or  $n_5$ ) indicates the data output from node  $x$  to node  $y$  at the  $n$ th iteration, and  $f_{xy}[\ ]$  stands for the function used to calculate  $xy(n)$ .

Based on the above initial data, only nodes  $n_1$  and  $n_2$  can be executed.  $n_1$  executes by consuming  $n_4n_1(0)$  and  $n_5n_1(0)$  to produce data  $n_1n_2(1)$  and  $n_1n_3(1)$ . Similarly,  $n_2$  executes to produce data  $n_2n_4(1)$ . Now  $n_3$  can execute to produce  $n_3n_4(1)$ , which in turn, along with the data  $n_2n_4(1)$ , enables  $n_4$  to execute. Thus a node is executable as soon as all its input edges have data. After execution, it produces data for all of its output edges. These node execution semantics are similar to those of transition firing in a Petri net [18], where a transition can fire, if and only if, all of its input places have tokens, and once a transition is fired, all of its output places get tokens. Thus, one can consider delay elements corresponding to the initial data can be viewed as the initial marking of the equivalent PN. When a data arrives at the input edge of a node and the node cannot execute because the data at the other input edges are not yet available, we say that the newly arriving data is *waiting* at the input edge of the node.

Note that in the above first iteration,  $n_3$  cannot execute until  $n_1$  finishes its execution. Thus there is a precedence relationship between  $n_1$  and  $n_3$  within the same iteration – intra-iteration precedence. Continuing the above execution, eventually,  $n_5$  and  $n_4$  execute, respectively, to produce input data for  $n_1$  to initiate the next iteration. This kind of precedence relationship between  $n_5$  (or  $n_4$ ) and  $n_1$  is called an *inter-iteration* relationship because

$n_5$  and  $n_4$  execute in the first iteration and  $n_1$  executes in the second iteration. Note that from the above initial data,  $n_1$  and  $n_2$  can execute at the same time by allocating one processor to each node. Thus within one iteration, multiple processors can be assigned to nodes without precedence constraints, within the help of the acrylic precedence graph (APG) of the RDFG obtained by deleting all edges having delay elements. This assignment by identifying the critical path is called an *admissible multiprocessor schedule* [2].

Consider another set of initial data (or initial markings)  $n_2n_4(0)$ ,  $n_3n_4(0)$ , and  $n_1n_2(0)$  (which can be obtained from the above first set of initial data by executing  $n_1$ ,  $n_2$  and  $n_3$ , thereby shifting delay elements from  $n_5n_1$  and  $n_4n_1$  to  $n_2n_4$  and  $n_3n_4$ , respectively). This results in a different schedule and a different APG (again by deleting the edges having delays) and may produce a shorter iteration period. The action of shifting delays to produce a different set of initial data is called *retiming* [2].

Because any edge in a RDFG has only two end nodes, any place in the corresponding PN model has exactly one input and one output transition. Thus it is decision-free or a marked graph (MG) [18]. The following theorem shows that if we execute transitions in the MG in an *ASAP* (as soon as possible) fashion, eventually, a steady state will be reached, where the average time period (called the cycle time in [18]) between two successive firings of any transition equals the maximum loop bound or iteration bound.

**Definition** [18]: Let  $S_i(f_i)$  be the time at which node  $n_i$  initiates its  $f_i$ th execution. The cycle time or iteration bound,  $C_i (=I)$ , of transition  $S_i$  is defined as

$$C_i = \lim_{f_i \rightarrow \infty} \frac{S_i(f_i)}{f_i}. \quad (1)$$

**Theorem 2**[18]: For a decision-free PN:

- (1) The number of tokens in a loop remains the same after any firing sequence.
- (2) All transitions have the same cycle time.
- (3) The minimum cycle time (maximum performance)  $C$  is given by

$$C = \max \left\{ \frac{T_k}{N_k}; k = 1, 2, \dots, q \right\}. \quad (2)$$

**Corollary 1:** As  $f_i$  approaches  $\infty$  or under static scheduling,  $S_i(f_i) = S_i^0 + Cf_i$ .

**Proof:** Since all transitions have the same cycle time  $C$ , from Eq. (1), we have  $S_i(f_i) = S_i^0 + Cf_i$ .  $\square$

The system is said to reach **steady state** if the above relation holds for every node. Consider an edge with no delay elements from  $n_i$  to  $n_j$ . Their execution moments  $S_i$  and  $S_j$  must satisfy the forward relation:  $S_j(f_j) \geq S_i(f_i) + r_i$ , where  $f_i = f_j$ . Thus, the two  $S$ 's are for the same iteration as its name (forward) implies. This relation comes directly from the firing rule. If there are  $x_i$  delay elements between  $n_i$  and  $n_j$ , then the above forward relation still applies except  $f_j = f_i + x_i$ . Now the two  $S$ 's are for different iterations; hence the resulting inequality derived from it will be called the backward relation. Using Corollary 1, we have  $S_j(f_j) = S_j(f_i) + x_i I$ , and the forward relation becomes  $S_j(f_i) \geq S_i(f_i) + r_i - x_i I$ . Hence we have:

**Lemma 2:** The following relation holds for FS scheduling:  $S_j(f_j) \geq S_i(f_i) + r_i - x_i I$  for an edge from  $n_i$  to  $n_j$  with  $x_i$  delay elements. This is, the  $f_j$ -th execution moment of  $n_j$  must be greater than that of  $n_i$  by an amount of  $r_i - x_i I$ .

Consider a path  $\Gamma$  from  $n_i$  to  $n_j$  with nonzero delay elements. The backward relation in [1] is  $t_j \geq t_i + \sum_{n_l \in \Gamma} (r_l - x_l I)$ , where  $t_i$  and  $t_j$  are start execution times of  $n_i$  and  $n_j$ , respectively. It is not clear in [1] whether  $t_i$  and  $t_j$  refer to the same iteration. As will be shown below, they are indeed for the same iteration. We rewrite the above backward relation:

$$S_j(f_j) \geq S_i(f_i) + \sum_{n_l \in \Gamma} (r_l - x_l I), \text{ where } f_i = f_j, \text{ which can be derived from Lemma 2 as follows.}$$

**Derivation of Backward Relation:** Consider a path with three successive nodes  $n_i - x_i - n_j - x_j - n_k$ . Applying Lemma 2 to the two arcs  $n_i - x_i - n_j$  (the arc from  $n_i$  to  $n_j$  with  $x_i$  delays) and  $n_j - x_j - n_k$ , respectively, we have

$$S_j(f_j) \geq S_i(f_i) + r_i - x_i I$$

and

$$S_k(f_k) \geq S_j(f_j) + r_j - x_j I.$$

Summing these two equations, we have

$$S_k(f_k) \geq S_i(f_i) + (r_i - x_i I) + (r_j - x_j I).$$

In general, for a path  $\Gamma$  from  $n_i$  to  $n_j$ , we have

$$S_j(f_j) \geq S_i(f_i) + \sum_{n_l \in \Gamma} (r_l - x_l I), \quad (3)$$

which is the backward relation.  $\square$

Note that the component in the sum of the right-hand side of the above inequality corresponds to the negative of the  $ij$  entries, i.e.,  $-q_{ij}^{(1)}$ , in the initial matrix. The backward relation for every pair of nodes, it satisfies the firing rule. Otherwise, the firing rule will be violated at some transition. Therefore, this is referred to as a **feasible scheduling**. To check whether a scheduling satisfies all backward relationships is quite time consuming. The following lemma helps to relieve this problem.

**Lemma 3:** if the  $f_i$ -th execution moment of  $n_j$  must be greater than that of  $n_i$  by an amount of  $-a_{ij}$ , i.e.,

$$S_j(f_j) \geq S_i(f_i) - a_{ij}, \quad (4)$$

then  $S_j(f_j)$  and  $S_i(f_i)$  satisfy the backward relation for any  $\Gamma$  from  $n_i$  to  $n_j$ .

**Proof:** This lemma holds due to the fact that  $-a_{ij}$  is the maximum of all  $\sum_{n_l \in \Gamma} (r_l - x_l I)$ .  $\square$

Thus we need to check only one inequality in Eq. (4) for each pair of nodes  $n_i$  and  $n_j$ . Eq. (4) will be used later to show that our fixed time scheduling satisfies the firing rule.

### 5.3 Scheduling Range

The inequality graph consists of the same nodes and edges as in the RDFG with edge lengths equal to the value on the right side of the above inequality. An example of the inequality graph of Fig. 1 is shown in Fig. 6. According to [1], the fixed left (right) limit of the scheduling range of any node  $n_j$  can be found by finding the longest path from the reference node  $n_r$  to  $n_j$  (vice versa). Using the Bellman-ford algorithm [20, 21], one can find the longest paths from  $n_r$  to all other nodes to find the left limits. By reversing all edges and multiplying all lengths by -1 in the inequality graph, one can find all longest paths to  $n_r$  to find correct limits [1].

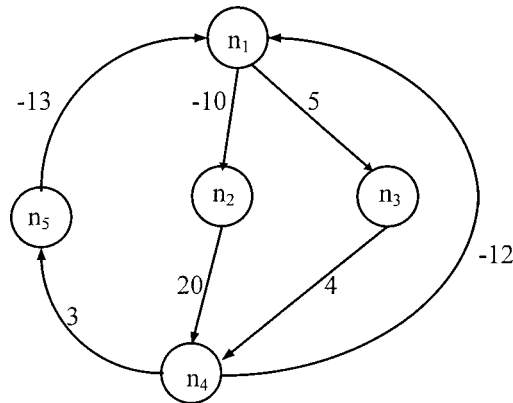


Fig. 6. The inequality graph of the DFG in Fig. 1.

**Theorem 3:** The scheduling range of a node  $n_j$  relative to  $n_r$  is 1.  $(0, I)$  if  $a_{rj} + a_{jr} > I$ ; 2. otherwise,  $(-a_{rj} \% I, a_{jr} \% I)$ .

**Proof:** Note that an edge length of the inequality graph corresponds to the negative value of an entry in the initial matrix; hence the longest path calculation corresponds to that of the shortest path in section 2. The final matrix with  $R = I$  contains the shortest distance between any two nodes. Thus the left limit equals  $-a_{rj}$ . The right limit corresponds to, according to [1], the negative of the longest distance from  $n_j$  to  $n_r$  in the inequality graph, which is  $a_{jr}$ . Since the scheduling is static, the scheduling times are periodic and should be expressed relative to the start time of each iteration; i.e., it should be within  $(0, I)$ . 1. If the magnitude of the range is greater than  $I$ , i.e.,  $a_{rj} + a_{jr} > I$ , then all possible scheduling times cover the entire range of one iteration period; hence the scheduling range is  $(0, I)$ . Otherwise, they cover only a portion of one iteration period, so we can use the module operation safely (unlike case 1), and the scheduling range is  $(-a_{rj} \% I, a_{jr} \% I)$ .  $\square$

Note that the time complexity of calculating the scheduling range based on the above explicit formula is  $O(n)$ . The following corollaries follow directly from Theorem 3.

Another alternative way to derive the scheduling range is as follows. There are a number of paths from  $n_i$  to  $n_j$ . The smallest  $S_j(f_i)$  which satisfies the inequality in Eq. (3) equals  $S_i(f_i) + \max_{n_l \in \Gamma} \sum (r_l - x_l I)$ . If we take  $n_i$  as the reference node  $n_r$  and  $S_i(f_i)$  as the start time of the  $f_i$ -th iteration of  $n_i$ , then the earliest time when  $n_j$  can execute in the  $f_i$ -th iteration relative to  $n_i$  is  $\max_{n_l \in \Gamma} \sum (r_l - x_l I) = -a_{ij} = -a_{rj}$ . On the other hand, if we take  $n_j$  as  $n_r$ , then the largest  $S_i(f_i)$  which satisfies the above inequality equals  $S_j(f_i) - \max_{n_l \in \Gamma} \sum (r_l - x_l I)$ . Since  $n_j$  is the reference node  $n_r$ , we take  $S_j(f_i)$  as the starting time of the  $f_i$ -th iteration of  $n_j$ ; then the latest time when  $n_i$  can execute in the  $f_i$ -th iteration relative to  $n_j$  is  $-\max_{n_l \in \Gamma} \sum (r_l - x_l I) = a_{ij} = a_{ir}$ . Thus the scheduling range of  $n_k$  during each iteration with respect to  $n_r$  is  $(-a_{rk}, a_{kr})$ .  $a_{kr}$  and  $-a_{rk}$  can be obtained from the final matrix. Thus, we need not construct the inequality graph to find the scheduling range.

The above derivation also derives the scheduling time of each node relative to  $n_r$  for two cases corresponding to the two limits  $(-a_{rk}$  and  $a_{kr})$  of the scheduling range (see Lemma 4). As a result, there is no need to schedule updates. They are called the **steady state ASAP and ALAP** schedulings. Under steady state, any  $n_j$  executes at time  $-a_{rj}(a_{jr})$  relative to  $n_r$  in an as soon (late) as possible fashion for *ASAP* (*ALAP*) scheduling. Hence we have:

**Lemma 4:** The *steady state ASAP and ALAP* schedulings are two feasible schedulings; i.e., they satisfy the firing rule.

**Proof:** This is true because they satisfy Eq. (4) and they are two feasible schedulings.  $\square$

**Corollary 2:** The mobility of a node  $n_j$  is not smaller than  $a_{jj}$ .

**Proof:** This mobility of  $n_j$  equals the length of the scheduling range, i.e.,  $(a_{rj} + a_{jr}) \geq a_{jj} \geq 0$ .  $\square$

The following corollary is obvious.

**Corollary 3:** If  $n_r$  is on the subcritical or critical loop through  $n_j$ , the mobility of  $n_j$  equals  $a_{jj}$ .

**Example:** Pick  $r = 2$  in Fig. 1. The scheduling ranges of nodes  $n_1$  to  $n_5$  are  $(10, 10)$ ,  $(0, 0)$ ,  $(15\%15, 16\%15)$ ,  $(20\%15, 20\%15)$  and  $(23\%15, 23\%15)$ , respectively, during each iteration which are consistent with both Figs. 4 and 5. Our CAD X-Window tool for DFG design automatically displays these scheduling ranges as shown in Fig. 7(a). We improve over that in [1] by displaying the ranges of each side on two sides and thickening all the edges on the critical loop.

Note also that if there are no directed paths from (to)  $n_j$  to (from) the reference node  $n_r$ , and that the lower (upper) bound of the scheduling range is  $-\infty(\infty)$  even though  $n_j$  is in certain loops. All nodes in the same loops will have the same unbounded scheduling ranges. This, however, poses no danger as long as any node's schedule is fixed, the schedules of the remaining nodes in the same loop will be updated to have bounded ranges. Updating the scheduling ranges will be considered in the next section.

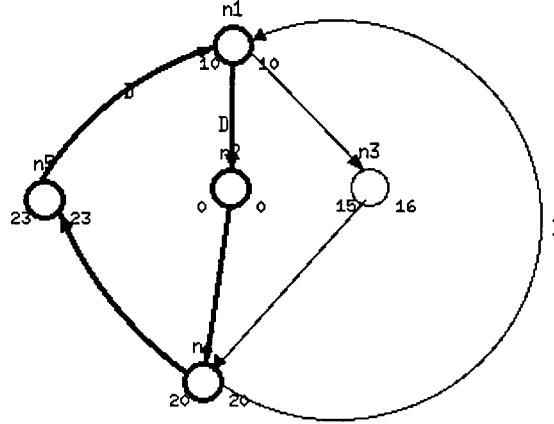


Fig. 7(a). The performance calculation of the DFG in Fig. 1.

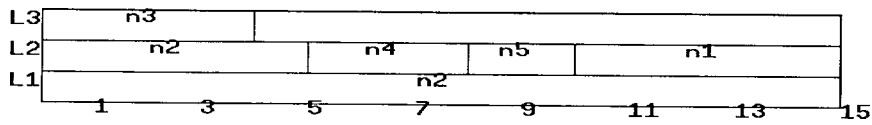


Fig. 7(b). Level diagram.

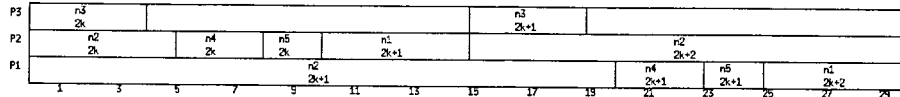


Fig. 7(c). Processor diagram, where “2k”, “2k+1”, and “2k+2” denote the iteration numbers.

### 5.4 Updating the Scheduling Range

In [1], the finalization of each node’s time scheduling entails updating the scheduling ranges of all nodes that have not been scheduled by constructing the inequality graph. It was shown in the last subsection that we can eliminate the construction with less time complexity. This subsection will show that there is no need for updates of scheduling ranges; the time complexity is, therefore, greatly reduced.

When the execution of  $n_e$  is postponed by an amount  $\delta$  from its earliest scheduling time, this affects the scheduling range of nodes yet to be scheduled. Equivalently, we can produce the same changes if we increase the execution time of  $n_e$  by  $\delta$  and execute it first. *The distance from  $n_c$  to  $n_d$  via  $n_e$  is decreased by  $\delta$ .* Since we calculate the scheduling range based on the final matrix, it is necessary to find the updated final matrix reflecting this increase. Let  $M^c$  and  $M^n$  denote the current and new updated matrices, respectively.  $M^n$  is calculated from  $M^c$  using the following lemma.

**Lemma 5:** (1)  $a_{cd}^n = a_{cd}^c$  if both  $n_c$  and  $n_d$  have been scheduled, where  $a_{cd}^n$  and  $a_{cd}^c$  denote entries in  $M^n$  and  $M^c$ , respectively. (2)  $a_{cd}^n = \min(a_{cd}^c, a_{ce}^c + a_{ed}^c - \delta)$  otherwise, where  $n_e$  is an arbitrary node.

**Table 1. Comparison between ART [19] and FIT for 4 filters.**

Example	I	PEs	ART	[19]	FIT
			Tcpu	Tcpu	Tcpu
Second-order section	3	4	0.01s	0.09s	<0.01s
4th-order Jaumann filter	16	3	0.03s	0.17s	<0.01s
4th-order All-pole filter	14	3	0.01s	0.13s	0.01s
5th-order	16	4	0.1s	0.33s	0.01s

**Table 2. Comparison between ART, [19] and FIT for 9 filters.**

Example	I	ART		[19]	FIT	
		PEs	Tcpu	PEs	PEs	Tcpu
2nd orthogonal filter	8	4	0.02s	5	4	0.01s
6th orthogonal filter	9	14	1.05s	16	14	0.01s
2nd Gray-Markel filter	7	3	0.01s	3	3	<0.01s
6nd Gray-Markel filter	7	8	0.21s	8	8	0.01s
2nd normalized filter	6	5	0.04s	5	5	0.01s
6nd normalized filter	6	14	0.49s	14	14	0.01s
2nd LMS filter	7	2	0.01s	2	2	<0.01s
6nd LMS filter	9	6	0.05s	6	6	<0.01s
4th Jaumann wave filter	8	3	0.03s	3	3	0.01s

**Table 3. Comparison between ART and FIT for benchmarks with large DFGs.**

Example	# nodes	# edges	I	ART		FIT	
				PEs	Tcpu	PEs	Tcpu
40th orthogonal filter	473	590	9	93	665.63s	91	0.09s
48th Gray-Markel filter	289	424	7	62	161.45s	62	0.05s
42nd normalized filter	337	461	6	96	235.24s	94	0.06s
32nd LMS filter	129	247	11	24	9.01s	24	0.03s

**Proof:** The distance  $a_{ej}$  from  $n_e$  to another node  $n_d$  is decreased by  $\delta$ , i.e.,  $a_{ed}^n = a_{ed}^c - \delta$ . (1) comes from the fact that the postponement cannot be arbitrarily long; it is limited by the scheduling algorithm so that the critical path between nodes already scheduled is not altered. (2) comes from the fact that the shortest distance between any two nodes  $n_c$  and  $n_d$  is the minimum of  $a_{cd}^c$  and  $(a_{ce}^c + a_{ed}^c - \delta)$ .  $\square$

Again, we do not need to construct the inequality graph to update the scheduling range. The time complexity of updating the final matrix based on the above explicit formula is  $O(n^2)$  since there are  $O(n^2)$  pairs of nodes. However, only entries  $a_{ij}$  and  $a_{ji}$  (where  $i$  is fixed and  $j$  ranges from 1 to  $n$ ) are needed to update the scheduling range according to Theorem 3. Thus, the time complexity of updating the scheduling range is  $O(n)$  compared with  $O(ne)$  using the inequality graph. Our fast scheduling algorithm requires no or very few updates, and the total time complexity for updating is  $O(n)$ .

## 6. TIME SCHEDULING WITHOUT UPDATES AND ELIMINATION OF TRANSIENT PERIODS

Whenever a node's time scheduling is fixed, the scheduling ranges of all nodes yet to be scheduled must be updated. The time complexity can be much reduced if we can determine the time schedulings of all nodes at once from the final matrix. This eliminates the above updates and the associated time complexity.

Experimental (Tables 1-3) results show that among all 17 benchmarks, only three (4th-order All-pole, 5th-order elliptic, and 4th Janumann wave filters) perform the worse; the remaining scheduling without updates achieves good performance. Further, after we finish processor assignment, we can adjust the scheduling of some nodes to reduce the number of processors. Most likely, only a few scheduling updates will be performed rather than all  $n$  updates in [1]. In order to be static, the fixed time scheduling must be extended to all, including the first, iterations. And it cannot be arbitrarily chosen due to the following potential problem: the earliest node to start execution in the first iteration may not be initially executable. If node executions are performed starting from the initially executable nodes, then there may be an initial transient period.

In the sequel, we will show that *steady state ASAP and ALAP* schedulings are two static rate-optimal time schedulings that need no updates and do not suffer from an initial transient period.

If *ASAP* scheduling is used (but not the steady state one), there will be an initial transient period (Fig. 4) when the start execution of certain nodes may not be fixed during each iteration. In other words, the scheduling is not static and incurs run-time overheads. Static scheduling fixes the start execution during each period and does not have to examine whether the node is executable or not. This section describes how the initial transient periods can be avoided so that the execution of every node takes a time period of  $I$  after its execution using the final matrix.

One solution is to extend the steady state scheduling to the first iteration. This implies that  $n_r$  executes at  $t = 0$ . This poses two problems. First,  $n_r$ , if arbitrarily chosen, may not be initially executable. Second, it may not be the earliest node to start execution in the first execution. How to pick  $n_r$  so that it is initially executable and is executed at  $t = 0$  so that steady state persists from the first iteration remains a research problem.

The *steady state ASAP and ALAP* schedulings can, however, be extended to the first iteration without the above problems as shown in the following theorem.

**Theorem 4:** For a strongly connected DFG with an arbitrary  $n_r$ , using the *Fully Static ALAP* (*FSASAP*) scheduling under no transient, (1) the earliest node for execution is node  $n_m$  with the minimum  $a_{mr}$  (maximum  $a_{rm}$ ), (2) the initial execution time  $H_k$  of any other node  $n_k$  is  $H_k = S_k^0 = a_{kr} - a_{mr}(-a_{rk} + a_{rm})$ , and (3) the execution time of any node  $n_k$  relative to  $n_m$  in each iteration is  $\phi_k = H_k \% I$ , where % is the “module” operator.

**Proof:** (1)&(2) can be proved by showing that (i) the scheduling is feasible (i.e., consistent with Eq. (4)) and (ii) the earliest node to execute in the first iteration is initially executable.

- (i) For any two nodes  $n_k$  and  $n_i$ ,  $a_{ki} + a_{ir} \geq a_{kr}$  ( $a_{ik} + a_{ri} \geq a_{rk}$ ), or  $a_{ir} - a_{mr} \geq a_{kr} - a_{mr} - a_{ki}$  ( $a_{ri} - a_{rm} \geq a_{rk} - a_{rm} - a_{ik}$ ). Now since  $S_k^0 = a_{kr} - a_{mr}(-a_{rk} + a_{rm})$  from (2), we have  $S_i^0 \geq S_k^0 - a_{ki}$  ( $S_k^0 \geq S_i^0 - a_{ik}$ ). Since the scheduling is static,  $S_i(f) = S_i^0 + fI$  and  $S_k(f) = S_k^0 + fI$ . Thus, Eq. (4) is satisfied at any iteration.
- (ii) Assume the contrary; then  $n_m$  is not initially executable, so that one of its input edges has no delay elements. Let the input node of the edge be  $n_{m'}$ ; then  $a_{m'r} < a_{mr} - r_{m'}$  ( $a_{rm'} > a_{rm} + r_{m'}$ ). Since the node execution time  $r_{m'}$  is non-negative, we have  $a_{m'r} (a_{rm'}) \leq a_{mr} (\geq a_{rm})$ ; then  $a_{mr} (a_{rm})$  is not the minimum (maximum), which violates the condition in (1).

(3) comes from the fact that there are no transient periods; the initial execution time  $S_k^0 \% I$  relative to  $n_m$  must remain the same for all iterations.  $\square$

Note that if  $k = m$ , then  $S_m^0 \leq a_{mr} - a_{mr} = 0$ , which is consistent with the fact that  $n_m$  is executed at  $t = 0$ . Also note that  $S_k^0 = a_{kr} - a_{mr}(-a_{rk} + a_{rm}) > 0$  for  $k \neq m$ .

**Example (FSALAP scheduling):** It is interesting to note that independent of the value of  $r$ ,  $a_{2r}$  is the minimum among all entries on column  $r$ . Pick an arbitrary  $r = 1$ . Hence  $m = 2$ , and  $n_2$  executes at  $t = 0$ . The first or initial execution time  $H$ 's of other nodes are as follows:  $n_1$  at  $t = (a_{21} - a_{11}) = 10$ ,  $n_3$  at  $t = a_{31} - a_{21} = 16$ ,  $n_4$  at  $t = a_{41} - a_{21} = 10 + 10 = 20$ , and  $n_5$  at  $t = a_{51} - a_{21} = 13 + 10 = 23$ . These  $H$ 's are consistent with those in Fig. 5. Since the scheduling is static, the start execution times of these nodes in each iteration can be obtained as  $H \% I$ .

Note that the reference node must be carefully chosen for non-strongly-connected DFGs as seen in Theorem 4. In the *FSALAP* (*FSASAP*) case, there must be directed paths from (to)  $n_r$  and  $n_m$  to (from) any other node  $n_i$  in the DFG; i.e., both  $a_{ri}$  and  $a_{mi}$  ( $a_{ir}$  and  $a_{im}$ ) are finite. Otherwise, some nodes' initial execution times will be  $\infty$ , which will happen when there are multiple nodes, defined as **input (output)** nodes, without input (output) arcs. In such cases, we can always construct an artificial input (output) node and connect arcs from (to) to (from) all input (output) nodes. In the implementation, if there is only one output node, we construct the *FSALAP* scheduling using the output node as  $n_r$ ; else if there is only one input node which may be artificial, we construct the *FSALAP* scheduling using the output node as  $n_r$ ; else if all nodes are in some loops, we select  $n_r$  based on Theorem 4.

These two schedulings are static and rate-optimal. Based on this fact, we simplify the scheduling algorithm in [1] in the next section. Our CAD tool is capable of handling both *FSASAP* and *FSALAP* scheduling to avoid the case where some  $S_i^0 = \infty$ .

## 7. PROCESSOR SCHEDULING

After the scheduling times of each node are determined, we need to assign processors to all nodes to execute the corresponding tasks. This processor assignment is based on a level diagram. A level diagram is a rectangular box with a horizontal axis from  $t = 0$  to  $t = I$  and a vertical axis indicating the levels. The major steps in processor scheduling are as follows:

- (1) Acquire the fix-time scheduling based on Theorem 4.
- (2) Construct the level diagram.
- (3) Perform processor assignment based on the level diagram.

Step 1 has linear time complexity of  $O(n)$ . Steps 2 and 3 also have a linear time complexity of  $O(n)$ . The details of steps 2 and 3 are discussed below. We will show that each substep has a linear time complexity of  $O(n)$ .

### 7.1 Level Diagram

After the time scheduling of each node is fixed, we can follow [1] and pack the nodes into a few levels as possible to obtain the level diagram (Fig. 7(b)) which will guide processor assignment. We classify the nodes into two categories: (a) those in some loops and (b) those not in any loops. We further classify (a) into: (a.1) those in mutually non-touching critical loops (called "c" type, having no common nodes with other "c" type critical loops), (a.2) those in critical loops having common nodes with some "c" type critical loops (called "b" type critical loops), and (a.3) those in subcritical loops ("s" type). The level diagram is constructed in the following order: (a.1), (a.2), (a.3) and (b). For the DFG in Fig. 8(a), type "c" critical loops include two loops: 1.  $n_1-n_2-n_3-n_4-n_5-n_7$  and 2.  $n_8-n_9-n_{10}-n_{12}-n_{13}-n_{15}$ . They are type (a.1) nodes.  $n_6$  and  $n_{11}$  are type (a.2) nodes in a type "b" critical loop:  $n_4-n_5-n_6-n_8-n_9-n_{11}$ .  $n_{14}$  is the only type (b) node. The above nodes cover all nodes, and there are no (a.3) nodes.

For case (a.1), the processor scheduling is simplified since we can just place successive nodes on the critical loop one after another. Hence, the time complexity is linear in the total number of nodes involved. For all nodes on the same critical loop, we need not fix their scheduling times or the associated scheduling updates. We observe that in most cases, the fraction of nodes that are on critical loops is over one-half of all the nodes in the DFG. For Figs. 7 and 8, the fractions are both 80%. Thus we have many fewer nodes to deal with in the level diagram (only one and 3 nodes, respectively for Figs. 7 and 8). Minimization of the number of processors can be more readily achieved.

Note that a node may appear in more than one level in the level diagram ( $n_2$  in Fig. 7(b)). In [1], processor assignment rearranged the level diagram so that no node could appear in more than one processor. This effectively precludes in [1] consideration of cases in which fractional iteration bounds and node execution time are greater than  $I$ . For instance, in Fig. 1, the execution time of  $n_2$  is 20 greater than  $I = 15$ . There is no way to achieve the minimum iteration period of  $I$  with  $n_2$  assigned to only one processor. In fact, in both Figs. 4 and 5, there are two processors assigned to  $n_2$ . In each processor, the iteration period is  $2I$ ; however, the effective iteration period as a whole is  $I$  since the throughput is doubled.

We, however, allow a node to appear in more than one level. If a node's execution time is greater than  $I$  and appears in  $k$  levels, we assign  $k$  processors to the node, and each such processor is assigned the same set of nodes that appear in the above  $k$  levels. This simplifies

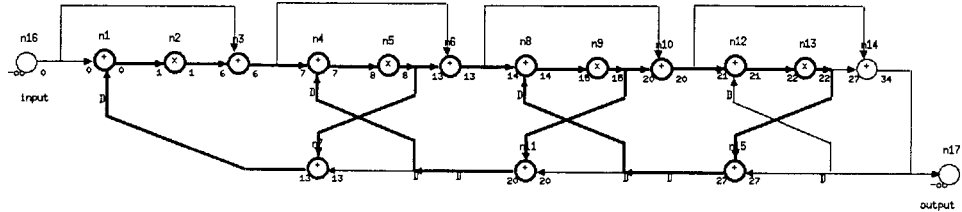


Fig. 8(a). The four-order all-pole lattice filter benchmark.

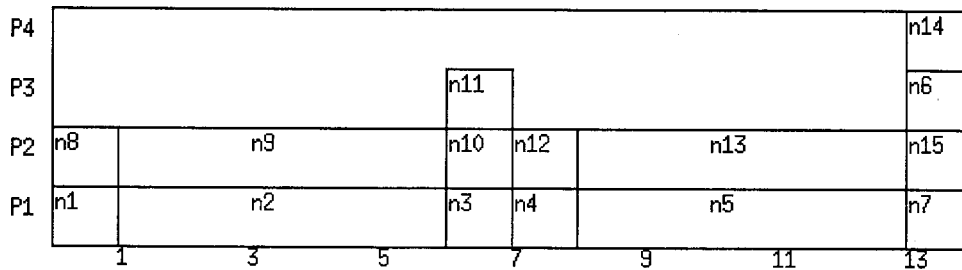


Fig. 8(b). Level diagram for fixed time scheduling.

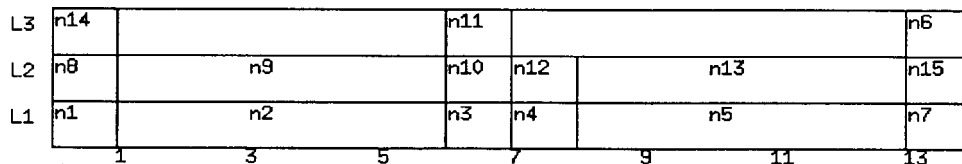


Fig. 8(b.1) Level diagram after timing adjustment.

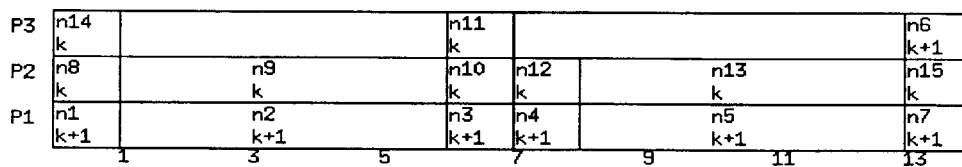


Fig. 8(c). Processor diagram, where “k” and “k+1” denote the iteration numbers.

processor assignment compared to that in [1]. Since the loop bound of a critical loop equals  $I$ , processors dedicated to nodes on a “c” type critical loop may achieve 100% utilization. Thus we should pack all nodes in a “c” type critical loop into an integral number of levels.

For case (a.2), 100% utilization is not possible. Hence, it is not necessary to pack the nodes into an integral number of levels, and there exist empty holes in the corresponding levels. To achieve better processor utilization, we attempt later to fill these holes with nodes on “b” or “s” type loops. Thus nodes on a loop tend to scatter slightly in the level diagram. To reduce the search time for a hole which fits a node, each time,  $t$  in the level diagram is assigned a variable  $level[t]$  to indicate the level into which the next node  $n_i$  with  $\phi_i = t$  should try to fit. Recall that  $\phi_i$  is the steady state start execution time for  $n_i$ . For  $n_6$  and  $n_{11}$  in Fig. 8 (a),  $level[13] = 3$ , respectively.

Thus when we schedule a node  $n_i$ , first we find out if an empty hole exists which fits  $n_i$  in level  $[\phi[n_i]]$ . If not, we increment level  $[\phi[n_i]]$  by one and repeat the above “find” process until the answer is “yes”. The trade-off is that an internal hole may leave unoccupied while a new level is created to hold the node. All the benchmark results indicated that this does not increase the final number of processors required.

The time complexity of this step is determined by the following two factors:

- (1) Searching an empty hole at level  $[\phi[n_i]]$  larger than the node computation time. This takes constant time complexity since the node computation time depends on the type of node (maximum of 5 for nodes of the multiplication type among all the benchmarks containing only two types of nodes: adder and multiplier) and does not grow with  $n$ .
- (2) If step 1 is not successful, then we search each level higher than level  $[\phi[n_i]]$  to make step 1 successful. Let  $\rho$  be the total number of levels. Because we do not start from level 0 in every search, the total time complexity is  $n + \rho \approx n$  since  $n \gg \rho$  (rather than  $n\rho$ ).

Because we start with an initial trial, the number of levels may not be a minima. We can reduce it in two ways as described in subsections 7.2 and 7.3, respectively.

## 7.2 Optimization

After scheduling subcritical loops, we try to eliminate the last few levels since their nodes have relatively large mobility using the procedure `optimize()`, which is the only place where we consider updates of scheduling ranges (see Lemmas 6 and 7). However, we are able to completely eliminate the updates as seen from the comment immediately following Lemma 7. We reschedule only segments purely in those subcritical loop scheduled last but not in other loops scheduled earlier; the two end nodes of the segment have been scheduled earlier. Rescheduling is performed under the condition that these two nodes’ schedules are not perturbed. As a result, those remaining nodes not in the segment are also not perturbed, and scheduling updates are minimized. The above segment is called a **pure segment** and is formally defined below:

**Definition:** Upon scheduling a subcritical loop, a segment of nodes on the loop,  $n_s, n_1, n_2, \dots, n_k, n_e$ , is defined as a **pure** one if

- (1)  $n_i$  has not been scheduled for all  $i$  from 1 to  $k$ ;
- (2)  $n_s$  and  $n_e$  have been scheduled earlier in other subcritical loops;
- (3) there are no branches of paths from and into the segment; i.e.,  $n_j$  has only one output node  $n_{j+1}$  and only one input node  $n_{j-1}$  for  $j = 1, 2, \dots, k$ , where  $n_0 = n_s$  and  $n_{k+1} = n_e$ .

The above action minimizes the scheduling updates as after we schedule the first node in the segment; the remaining nodes in the segment may be continuously scheduled without scheduling updates of the remaining nodes not in the segment. Thus in most benchmarks, few or even none of the nodes will change their scheduling times, which are fixed by Theorem 4. n13-n14-n12 in Fig. 8(a) is a pure segment, where  $n_s = n_{13}$  and  $n_e = n_{12}$ . The corresponding level diagram for fixed-time scheduling is shown in Fig. 8(b) and requires 4 levels (hence 4 processors). It can be reduced to 3 levels (Fig. 8(c)) by rescheduling  $n_{14}$ . The theory is developed as follows.

In the sequel, we assume **FSASAP** scheduling. The case of **FSALAP** can be similarly treated. We reschedule the segment nodes in increasing order of  $j = 1$  to  $k$ .

**Lemma 6:** For **FSASAP**, upon rescheduling  $n_1$  of a **pure segment**, in order not to change the scheduling of any node not in the segment that has been scheduled, the scheduling range of  $n_1$  must be  $(-a_{r_1}\%I, (-a_{r_1} + \Delta_1)\%I)$ , where  $\Delta_1 = a_{r_1} + a_{1e} - a_{re}$  is the mobility of  $n_1$ , defined as the magnitude of its scheduling range.

**Proof:** From Theorem 4, the earliest time that  $n_1$  can execute is  $-a_{r_1}\%I$ . Since the scheduling time of  $n_e$  is fixed at  $-a_{re}$ , the latest scheduling time of  $n_1$  is  $(-a_{re} + a_{1e})\%I = (-a_{r_1} + \Delta_1)\%I$ . This is to allow nodes  $n_1, n_2, \dots, n_e$  to fire in a successive fashion; i.e.,  $n_j$  fires as soon as its previous node on the pure segment  $n_{j-1}$  finishes execution.  $\square$

Note that  $\Delta_1$  equals the difference between the two distances from  $n_s$  to  $n_e$ : one the shortest and the other for the path corresponding to the pure segment. For the example in Fig. 8(a), the scheduling range for  $n_{14}$  with the scheduling of  $n_{s=13}$  and  $n_{e=12}$  fixed is (13, 6). Thus, we can adjust its scheduling of  $n_{14}$  from  $\phi = 13$  to 0 (i.e.,  $\delta_1 = 1$  to eliminate level 4, and there is no need to update the schedulings of other nodes.

Note that the  $15 \times 15$  final matrix is too large to display in Fig. 8. However, the scheduling range for  $n_{14}$  can be inferred from Fig. 8(b) since  $n_{14}$  must execute after  $n_{13}$  or before  $n_{12}$ . The formula in the above lemma is useful for CAD implementation.

Let  $\Delta_j$  and  $\delta_j$  be the mobility and postponement (i.e., the amount of shift away from static scheduling) of node  $n_j$ , respectively. The processor bound is defined as the sum of node computation times divided by the iteration bound; i.e.,  $P_b = \frac{T}{I}$ .

After we move  $n_1$  to a lower level, if the total number of levels is still less than  $P_b$ , we may repeat the above rescheduling process for  $n_2$ , and  $n_3$  and so on. Note that the earliest scheduling times for nodes from  $n_2$  to  $n_k$  have now been increased by  $\delta_1$ . They have the same scheduling range and hence the same mobility, which are both reduced by  $\delta_1$ . After we rescheduled  $n_2$ , the earliest scheduling times for nodes from  $n_3$  to  $n_k$  will be increased by  $\delta_2$ . And their scheduling range and mobility will both be reduced by  $\delta_2$ . Also for any  $j \in \{2, 3, \dots, k\}$ , its latest scheduling time remains unchanged. This is because it is determined by the length of the path (on the pure segment) from  $n_j$  to  $n_e$  and none of the nodes  $n_i$  ( $k \geq 1 > j$ ) have been rescheduled. The above discussion leads to the following lemma.

**Lemma 7:** For **FSASAP**, upon rescheduling  $n_j$  of a **pure segment** (after all  $n_k, k = 1, 2, \dots, j-1$  have been rescheduled) in order not to change the scheduling of any node not in the segment that has been scheduled, (1) the scheduling range of  $n_j$  must be

$$((-a_{r_j} + \sum_{i=1}^{j-1} \delta_i)\%I, (-a_{re} + a_{je})\%I) \text{ and (2) } \Delta_j = \Delta_1 - \sum_{i=1}^{j-1} \delta_i.$$

**Proof:** (1) Recall that the postponement of a node is equivalent to adding the postpone to the node computation time. Hence the shortest distance from  $n_s$  to  $n_j$  is decreased by  $\sum_{i=1}^{j-1} \delta_i$ , the corresponding sum of the postponements of nodes. Similar to the proof of Lemma 6, the earliest scheduling time of  $n_j$  equals  $(-a_{r_j} + \sum_{i=1}^{j-1} \delta_i)\%I$ . The latest scheduling time of  $n_j$  is solely determined by  $H_e$ , which is not affected by the postponements. Hence, it is  $(-a_{re} + a_{je})\%I$ . (2) The mobility equals the magnitude of the scheduling range. Hence  $\Delta_j = -a_{re} + a_{je} - (-a_{r_j} + \sum_{i=1}^{j-1} \delta_i) = \Delta_1 - \sum_{i=1}^{j-1} \delta_i$ , where we have used the fact that prior to the rescheduling of the **pure segment**, the mobility of  $n_j$  equaled  $\Delta_1 = -a_{re} + a_{je} - (-a_{r_j})$ .  $\square$

Note that there is no need for scheduling range updates as long as we know the value of  $-a_{re} + a_{je}$ . This due to the fact that for each  $n_j$  ( $j$  from 1 to  $k$ ), its latest scheduling times is fixed at  $-a_{re} + a_{je}$ . Thus  $n_j$  can be rescheduled to any time in  $(0, I)$  as long as it is less than  $(-a_{re} + a_{je})\%I$ .

Let  $P_\beta$  be the subprocessor bound, the processor bound formed by nodes  $n_w$  in type “b” and “s” loops but not in type “c” loops.

The following procedure **pure\_segment\_H\_adjust()** performs the acitons in Lemmas 6 and 7.

```

pure_segment_H_adjust()
{
  for (i = 1; i <= k; i++)/*for nodes in the segment  $n_s n_1 n_2, \dots, n_k n_e$ */
  {
    find  $\Delta_i$ ;
    calculate the FSASAP scheduling time of  $n_i$  using Lemma 7;
    for (h = 0; h ≤ max(I,  $\Delta_i$ ); h++)/*max(I,  $\Delta_i$ ) is to limit the search in one iteration period*/
    {
      increase the scheduling time of  $n_i$  by h;
      Let  $x$  be the level which is completely filled
      (does not have any holes) and all the lower levels
      are also completely filled;
      for ( $j = P_\beta$ ;  $j \geq x$ ;  $j--$ )
      {
        search an empty hole large enough to hold  $n_i$ ;
        if found, break;
      }
      if found, break;
    }
    if (not found)
      return FALSE;
     $\delta_i = h$ ;
    move  $n_j$  to level  $j$  accordingly;
    return True;
  }
}

```

The pseduo code for procedure optimize() is as follows:

```

optimize()
{
  while (total number of levels >  $P_\beta$ )
  {
    select the subcritical loop with the largest slackness;
    for each pure segment
    {
      if(!pure_segment_H_adjust())
        return;
    }
  }
}

```

The worst time complexity is  $O(nI\rho)$ . However, as mentioned earlier, among all the benchmarks, only three need to execute **optimize()** to reduce the number of processors. For these cases, it is rather easy to fit pure segments into lower levels due to their large slackness and the large holes in the last few levels. In addition, experimental results indicate that for large benchmarks with many type “n” nodes, executing **optimize()** does not help to reduce the number of processors. This is because the fragmented holes can be most easily filled by type “n” nodes; hence there is no need to rearrange the holes and the nodes already in the level diagram to reduce the number of levels required. This helps to reduce the longer time required to process large benchmarks. As a result, the corresponding time complexity does not grow with  $n$ .

### 7.3 Scheduling of Type “n” Nodes

Second, we start to schedule nodes which are not in any loops (case (b)). We first sort and schedule them in order of decreasing node computation times. We then try to fit them into empty holes. Since the maximum types of nodes are limited (at most three in all benchmarks), an efficient (linear time complexity) sorting is to scan through the nodes and place each node in the queue for its type.

The start execution time of the nodes can be any times since its scheduling range is infinitely large. Hence we can fit the nodes into any internal holes large enough to achieve higher utilization for processors assigned to these levels. We use an eager approach to fill larger empty spaces first; hence the scheduling is in decreasing order of node execution times. Whenever we fit a node into a hole, we always insert the node into the left hand side of the hole to enlarge the region filled with nodes. This does not increase the number of holes. Otherwise, if the node is inserted in the middle of the hole, it will form two new smaller fragmented holes. This does not work well in the case of many fragmented holes too small to fit nodes. As a result, new levels must be created to hold these nodes. Since we cannot compact these fragmented holes to make larger holes due to the fixed time schedulings, the only way to enhance utilization is to mix type “n” nodes and others together in the same level. That is, we may squeeze out some case (a) nodes to higher levels to create larger holes. We then fit case (b) nodes continuously into them to create larger filled regions in the level and hence reduce fragmentation. This step takes linear time complexity.

### 7.4 Adjustment of Initial Scheduling

So far, we have determined the  $\phi$ 's of all the nodes in the DFG. To be complete and static, we have to recalculate the initiation (i.e., first) scheduling time  $H$  of each node. This is because according Theorem 4, the  $H$  of  $n_i$  is determined by the shortest distance from  $n_r$  to  $n_i$ . When we reschedule a node on the path from  $n_r$  to  $n_i$ , this may affect the corresponding shortest distance. Recall that we can shift the time reference to make all  $H \geq 0$ . Hence we fix the  $H$ 's for nodes in some loops and calculate the  $H$  for any type “n” node using the forward or backward relation. Afterwards, we then shift all  $H$ 's by the same amount to make all  $H \geq 0$ .

For those rescheduled type “n” nodes with no directed paths from (to) nodes in loops to (from) them, called **Region I(II) boundary nodes**, we can compute their  $H$ 's using the forward (backward) relation. In Fig. 8,  $n_{13}$  is a Region II boundary node. The later backward

relation implies that we do so only if the H's of all its output nodes have been determined. Consider the backward relation first; the forward one can be treated in a similar manner. The following procedure finds all Region I(II) boundary nodes:

```

boundary()
{
  find_type_n_nodes()
  {
    for each node  $n_i$ 
      if ( $a_{ii} = \infty$ ) then  $n_i$  is a type "n" node;
      then  $n_i$  is a type "n" node;
    }
  for each node  $n_j$  in a certain loop
  {
    if (exists an input type "n" node)
      declare  $n_j$  a Region I boundary node;
    if (exists an output type "n" node)
      declare  $n_j$  a Region II boundary node;
    }
  }
}

```

The time complexity of **boundary()** is linear in the total number of nodes.

The latest scheduling time of the initiation scheduling of such a node  $n_i$  equals

$\alpha_i = \min_{n_j \in O(n_i)} H(n_j) + a_{ij}$ , where  $O(n_i)$  is the set of output nodes of  $n_i$ . The scheduling range is, therefore,  $(-\infty, \alpha_i)$ . To match its  $\phi$ , we minimize the movement of  $H_i$  and choose  $H_i = \alpha_i - (\alpha_i \% I - \phi + I) \% I$  to maintain  $H_i \% I = \phi$ . After the H's for all type "n" nodes have been computed, some such H's may be negative. To make all the H's nonnegative, we select the node  $n_\lambda$  with the most negative H and increase all H's by an amount of  $-H_\lambda$ .  $n_i$  in Region I or II is said to be schedulable if its  $\alpha_i$  can be computed. This occurs when all the H's of its output nodes have been determined.

A naive way to adjust H for type "n" nodes is to search for any schedulable nodes and to then compute its H in the above manner. We continue this way until all the nodes' Hs have been determined. The corresponding time complexity is quadratic to the total number of type "n" nodes. A more efficient way is implemented in procedure **adjust\_H()**. A variable **temp** (initialized to a very large negative number) is assigned to each type "n" node. Whenever a node  $n_i$  is schedulable, we compute  $H(n_i) + a_{ij}$  for each of its input (consider the backward relation) nodes  $n_j$ . We assign **temp** = min (**temp**,  $H(n_i) + a_{ij}$ ). If all the output nodes of  $n_i$  have been executed, then  $H(n_i) = \text{temp}$ . The time complexity is linear to the number of edges involved. However, in VLSI, the maximum number of fan-ins and outs is limited. Hence it is also linear to the number of type "n" nodes:

```

adjust_H();
{
  assign tempI = +9999 for every type "n" Region I node;
  assign tempII = -9999 for every type "n" Region II node;
boundary();
  /*find all boundary nodes that are in loops and some of whose input or output
  nodes are type "n" nodes; */
}

```

```

make all boundary nodes schedulable;
for each schedulable node  $n_i$ 
{
  for each input node  $n_j$  whose H has been determined in this procedure
  {
    compute  $t = H(n_i) + a_{ij}$ ;
    tempII = min(tempI, t);
    if (the t for all output nodes of  $n_j$  have been computed)
      make  $n_j$  schedulable and assign  $H(n_j) = \text{temp}$ ;
  }
  for each output node  $n_j$  whose H has been determined in this procedure
  {
    compute  $t = H(n_i) - a_{ij}$ ;
    tempII = max(tempII, t);
    if (the t for all input nodes of  $n_j$  have been computed)
      make  $n_j$  schedulable and assign  $H(n_j) = \text{temp}$ ;
  }
  make  $n_i$  no longer schedulable;
}
}

```

### 7.5 Processor Assignment

Processor assignment is based on the level diagram. If all the nodes in a level do not appear in other levels, then we assign one processor to the level. Otherwise, a node in the level may appear in other levels. Note that this node will appear in consecutive levels. In this case, we can select a set  $\Gamma$  of consecutive, but as few as possible, levels so that it covers all nodes in  $\Gamma$  completely; i.e., each node in  $\Gamma$  appears in no other levels. We assign  $|\Gamma|$  processors, and all the nodes in  $|\Gamma|$  to each of these processors. The time complexity of this selection step is linear to the number of levels involved, which is less than  $n$ . Hence the total time complexity of the processor assignment stage is  $O(n)$ .

It is easy to see that the iteration period in each such processor is  $\Gamma I$ . The reason for choosing as few consecutive can see that the number of final processors equals the total number of levels in the level diagram.

### 7.6 Summary of Processor Scheduling

- (I) Time Scheduling: Construct an *FSASAP* or *FSALAP* time scheduling to find  $H$  and  $\phi$  for all nodes based on Theorem 4.
- (II) Level Assignment:
  - (1) For each type “c” loop, fit its nodes into an integral number of levels.
  - (2) Schedule type “b” or “s” loops in increasing order of slackness.
    - (2.1) For each node  $n_i$  in a selected loop,
      - (2.1.1) Find out if an empty hole exists to fit  $n_i$  at level  $[\phi[n_i]]$ .
      - (2.1.2) If not, increment level  $[\phi[n_i]]$  by one and repeat (2.1.1) until the answer is “yes”.
    - (2.2) if  $n < \text{large}$  (an adjustable constant, but 50 in our case), do **optimize()**.
  - (3) Sort the rest of the nodes (not in any loops) in order of decreasing execution time.

- (3.1) For each selected node  $n_i$ , select the lowest level  $j$  that has not been completely filled.
  - (3.2) Find an empty hole large enough to fit the node. Insert the node into the left hand side of the hole to enlarge the region filled with nodes.
  - (3.3) If (b) cannot be satisfied, i.e., it clashes with some nodes on the same level  $j$ , shift each such clashing node  $n_k$  on level  $j$  to a level higher than  $j$  with an empty hole large enough to fit the node. Do not change  $\phi_k$ .
  - (4) Do adjust\_H() to adjust initial scheduling;
- (III) Processor Assignment:
- (1) Select a set  $\Gamma$  of consecutive, but as few as possible, levels so that it covers all the nodes in  $\Gamma$  completely; i.e., each node in  $\Gamma$  appears in no other levels.
  - (2) Assign  $|\Gamma|$  processors and all nodes in  $|\Gamma|$  to each of these processors.
- The total time complexity is  $O(n)$  as each substep is the same time.

## 8. THE CAD TOOL (FIT) AND BENCHMARK RESULTS

In [14], we reported a CAD tool used to draw, edit, and simulate a DFG and to find critical and subcritical loops, and iteration bounds. Based on this CAD tool, we have developed and implemented a fast processor assignment algorithm to allow the designer to view graphically critical loops, scheduling ranges, and level and processor assignment diagrams.

FIT integrates the following functions:

- (1) DFG Draw and Edit capability including erase, move, copy, past, etc.;
- (2) function assignment to each node and the ability to simulate the whole DFG to verify against the specifications;
- (3) analysis of DFG performance and properties such as iteration bounds, critical loops, liveness, safeness, etc.;
- (4) application of our theory of the final matrix to the scheduling of DFG nodes among processors.

All the tools [1, 16, 17, 19, 22] reported deal only with

- processor assignment;
- single-rate DFGs (SRDFG); our CAD tool is able to handle both SRDFG and MRDFG (multiple-rate DFG) [23];
- textual data only; no graphical display of either DFGs or processor assignments is available without the printing of postscript files. In other words, no on-line graphical display is available; our CAD tool is able to handle both graphical and textual inputs (useful for very large DFGs);
- steady-state scheduling; initial transients were not discussed.

Often the designer draws a new, or starts with an existing, graphical DSP and then translates it into textual formats to be processed by the software. In the former case, several iterations may be required before the design can be finalized. A CAD tool that can erase, move and copy graphical objects as ours can will be very helpful in facilitating the drawing process.

*FIT* will also serve as a computer-aided education tool for DFG design. Students can understand the operation of DFG by simulating the DFG in a micro-fashion using the “Step” mode. Further, the tool is currently able to simulate a DFG with the “Trace” mode on to help students understand the meaning of iteration bounds, critical loops, static rate-optimal scheduling, and so on.

The above algorithm has been incorporated into our CAD tool to allow the designer to draw the DFG and click on buttons to view critical loops (in thicker lines), scheduling ranges, level diagrams and processor assignments. See Fig. 7(a)-(c), where the *FSASAP* scheduling is used. Due to the limited size of the display window, we show the iteration numbers underneath each node number. The number of iterations shown in Fig. 7(c) is two, barely large enough to cover node  $n_2$ , which has the longest node execution time. The iteration number of  $n_i$  relative to  $n_r$  is  $H_i/I$ . Note that among all five nodes, only one node is not on the critical loop, and that the level diagram can be trivially constructed. Thus, there is no need to adjust the level diagram to reduce the number of levels and hence the number of processors. This is indeed true for many benchmarks in which most nodes are in nontouching critical loops.

See Tables 1-3, where the program was run on a SUN Sparc 10 at 40 Mhz with 32Mb main memory. Table 1 and 2 show a comparison between ART [16], Barnwell’s result [19] and *FIT* for benchmarks with relatively smaller DFGs compared to those in Table 3. Both tables indicate that ART and *FIT* use the same number of processors, but fewer than [19]. *FIT* uses CPU time no longer than 0.01 second, which is shorter than either ART and [19]. Table 3 shows a comparison between ART and *FIT* for benchmarks with large DFGs. *FIT* uses fewer processors but runs much faster than ART. It is interesting to observe that for large benchmarks (Table 3), there is no need to perform `optimize()` to reduce the number of processors. This is because in such cases, there are many type “n” nodes to fill empty holes. The number of processors used in this case depends on how efficiently we pack type “n” nodes.

ART developed by Wang et al. [16] runs faster than all the above schemes except *FIT*. It minimizes the number of processors, and they developed a CAD tool called *ART*. The idea is that all fully static periodic schedules must be upper bounded by a *cutofftime*. Comparing all such schedules allows us to find one that minimizes the number of processor. This finite *cutofftime* also leads to faster scheduling as shown by their benchmark results. However, their worst case time complexity is  $O(n^2(d + e + 1) + nd)$ , smaller than  $O(n^3)$  used by [1] and our  $O(n)$ , where  $n(e)$  is the total number of nodes (edges).

The reason why *FIT* is fast is due to the fact that the *FSASAP* and *FSALAP* time schedules and scheduling ranges come directly from the final matrix while ART have to use graphs to redo some computations similar to those involved in the Final matrix. In addition, ART incurs extra overheads finding  $T_{cutoff}$  and within which to search for the optimum time schedule. [1], on the other hand, relies on repeated schedule updates to fix the time schedule. Further, we do not need to search very much for empty holes as we can continuously arrange the nodes on a critical loop in the same levels.

The reason why *FIT* uses fewer processors is due to the fact that we schedule nodes on a critical or subcritical loop continuously more or less in the same levels, which tends to reduce the number of fragmented holes. Further, we eliminate as many fragmented holes as possible using (type “n”) nodes not in any loops which have infinitely large scheduling

ranges and can be inserted into any place in the level diagram. If the fragmented holes are too small to fit type “n” nodes, we squeeze out the fragmented holes by shifting all but one non-“n” type node into higher levels, and fill the remaining empty region completely with type “n” nodes. This creates the largest continuous filled region and hence leads to higher utilization of the processor, resulting in fewer processors needed.

Most produce the optimal scheduling using our fixed-time scheduling without scheduling updates. Few cases require scheduling updates to further reduce the number of processors. An example of a four-order all-pole lattice filter benchmark is shown in Fig. 8. Note that nodes  $n_{16}$  and  $n_{17}$  are input and output nodes, respectively. They are created artificially, and their execution times are zero so that they will not be considered in processor assignment. Among the total of 15 nodes, 12 are in two nontouching critical loops and can be scheduled easily on two processors. The remaining 3 nodes can be assigned to two levels (Fig. 8(b)), which can be reduced to one by shifting forward the scheduling times of  $n_{14}$  by one time unit as shown in Fig. 8(c). Note that there is no need to update the scheduling ranges as in [1]. As a result, the time complexity is greatly reduced.

The time schedulings of other nodes are not affected. This is because the corresponding mobilities are among the largest. Note that the higher the level, the fewer the nodes; hence it takes less time to remove, if possible, the last level.

## 9. CONCLUSIONS

We have supplemented the scheduling theory of Heemstra de Groot et al. by showing that the final matrix is not only useful for finding the iteration bound and critical loop, but also useful to derive the explicit formulas for the slack time, the scheduling range and its update, and the initial scheduling to avoid transient periods. Thus, the theory presented in this paper eliminates some redundant steps (e.g., inequality graphs are no longer needed) from the scheduling algorithm of [1]. We have proved that both the *steady state* **ASAP** and **ALAP** schedulings satisfy the firing rule, and that they are static rate-optimal. Since the start execution times of all the nodes are fixed, we can simplify the scheduling algorithm in [1] by eliminating most of the scheduling updates. We further simplify the scheduling by scheduling critical loops ahead of subcritical loops since a major portion of the nodes are in nontouching critical loops. We have also considered the case of large node execution times, which was not considered in [1]. No unfolding is required as in [1] compared with [2], where unfolding takes extra time and space. However, we did not consider the case in which  $I$  is a fraction as in [2]. It is expected that with slight modification, our simplified scheduling algorithm can handle such fractional cases.

We have implemented the theory presented above into our earlier CAD tool, which can find iteration bounds, critical and subcritical loops. This results in a single tool, unlike [1] and [2], for DFG scheduling. Finally, we have enhanced our tool to handle PERT charts, which do not have any loops, and to perform simulations to verify both timing and functional behaviors.

**REFERENCES**

1. S. M. Heemstra de Groot, S. H. Gerez, and O. E. Hermann, "Range-chart-guided iterative data-flow graph scheduling," *Transactions on Circuits and Systems*, Vol. CAS-39, No. 5, 1992, pp. 351-364.
2. K. K. Parhi, "Algorithm transformation techniques for concurrent processors," *IEEE Proceedings*, Vol. 77, 1989, pp. 1879-1895.
3. S. M. Heemstra de Groot, "Scheduling techniques for iterative data-flow graphs, and approach based on the range chart," Ph.D. Dissertation, Unvi. Twente, Faculty of Electrical Engineering, Dec. 1990.
4. J. Blazewicz, "Selected topics in scheduling theory," *Surveys in Combinatorial Optimization*, P. L. Hammer, Ed Amsterdam, North-Holland, 1987, pp. 1-59.
5. W. A. Kohler, "A preliminary evaluation of the critical path method for scheduling tasks on multiprocessor systems," *IEEE Transactions on Computers*, Vol. C-24, 1975, pp. 235-238.
6. D. A. Schwartz, "Synchronous multiprocessor realizations of shift invariant flow graphs," Ph.D. Dissertation, Technical Report DSPL-85-2, Georgia Institute of Technology, July 1985.
7. S. M. Heemstra de Groot and O. E. Hermann, "Evaluation of some multiprocessor scheduling techniques of atomic operations for iterative DSP graphs," in *Proceedings of European Conference on Circuit Theory and Design*, 1989, pp. 400-404.
8. M. Renfors and Y. Neuvo, "Fast multiprocessor realization of digital filters," in *Proceedings of International Conference on Acoustics, Speech, Signal Processing*, 1989, pp. 916-919.
9. M. Renfors and Y. Neuvo, "The maximum sampling rate of digital filters under hardware speed constraints," *IEEE Transactions on Circuits and Systems*, Vol. CAS-28, No. 2, 1981, pp. 196-202.
10. A. Fettweis, "Realizability of digital filter networks," *IEEE Proceedings*, Vol. 77, No. 12, 1989, pp. 1879-1895.
11. D. A. Schwartz and T. P. Barnwell, "Cyclo-static multiprocessor scheduling on the optimal realization on shift invariant data flow graphs," in *Proceedings of International Conference on Acoustics, Speech, Signal Processing*, 1985, pp. 1384-1387.
12. Y. Yaw, B. Wei, C. V. Ramamoorthy, and W. T. Tsai, "Extensions on performance evaluation technique for concurrent systems," *International Computer Software and Application Conference*, 1988, pp. 480-484.
13. S. H. Gerez, S. M. Heemstra de Groot, and O. E. Hermann, "A polynomial time algorithm data flow graphs," *IEEE Transactions on Circuits and Systems*, CAS-I, Vol. 40, 1993, pp. 629-634.
14. D. Y. Chao and D. T. Wang, "Iteration bounds of single-rate data flow graphs for concurrent processing," *IEEE Transactions on VLSI*, Vol. 3, No. 3, 1995, pp. 393-403.
15. R. W. Floyd, "Algorithm 97: shortest path," *Communications of ACM*, Vol. 5, No. 6, 1962, pp. 345.
16. D. J. Wang and Y. H. Hu, "Multiprocessor implementation of real-time DSP algorithms," *IEEE Transactions on VLSI*, Vol. 3, No. 3, 1995, pp. 393-403.
17. L. G. Jeng and L. G. Chen, "Rate-optimal DSP synthesis by pipeline and minimum unfolding," *IEEE Transactions on VLSI*, Vol. 2, No. 1, 1994, pp. 81-87.

18. C. V. Ramamoorthy and G. S. Ho, "Performance evaluation of asynchronous concurrent systems using Petri nets," *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 5, 1980, pp. 440-449.
19. P. R. Gelabert and T. P. Barnwell III, "Optimal automatic periodic multiprocessor scheduler for fully specified flow graphs," *IEEE Transactions on Signal Processing*, Vol. 41, 1993, pp. 858-888.
20. R. Bellman, "On a routing problem," *Quarterly of Applied Mathematics*, Vol. 16, No. 1, 1958, pp. 87-90.
21. L. R. Ford, Jr. and S. M. Johnson, "A tournament problem," *The American Mathematical Monthly*, Vol. 66, 1959, pp. 387-389.
22. C. Y. Wang and K. K. Parhi, "Dedicated DSP architecture synthesis using the MARS design system," in *Proceedings of International Conference on Application of Special Array Processing*, 1992, pp. 21-36.
23. D. Y. Chao, "Performance of multi-rate data flow graphs for concurrent processing," *Journal of Information Science and Engineering*, Vol. 13, No. 1, 1997, pp. 85-123.



**Daniel Y. Chao (趙玉)** received the Ph.D. degree in electrical engineering and computer science from the University of California, Berkeley, in 1987. From 1987-1988, he worked at Bell Laboratories. In 1988, he joined the computer and information science department of New Jersey Institute. In 1994, he joined the MIS department of NCCU as an associate professor. Since February, 1997, he has been promoted to a full professor. His research interest was in the application of Petri nets to the design and synthesis of communication protocols. He is now working on CAD implementation of a multi-function Petri net graphic tool. He has published 77 (including 19 journal) papers in the areas of communication protocols, Petri nets, DQDB, networks, FMS, data flow graphs and neural networks.