

# Minimizing Resources in a Repeating Schedule for a Split-Node Data-Flow Graph

Timothy W. O'Neil  
Dept. of Computer Science and Engineering  
Univ. of Notre Dame, Notre Dame IN 46556  
toneil@cse.nd.edu

Edwin H.-M. Sha  
Department of Computer Science  
Univ. of Texas at Dallas, Richardson TX 75083  
edsha@utdallas.edu

## ABSTRACT

Many computation-intensive or recursive applications commonly found in digital signal processing and image processing applications can be represented by *data-flow graphs* (DFGs). In our previous work, we proposed a new technique, *extended retiming*, which can be combined with minimal unfolding to transform a DFG into one which is rate-optimal. The result, however, is a DFG with split nodes, a concise representation for pipelined schedules. This model and the extraction of the pipelined schedule it represents have heretofore not been explored. In this paper, we demonstrate one scheduling algorithm for such graphs, and then discuss a way to reduce the hardware requirements of the resulting schedule. In the process, we state and prove a tight upper bound on the minimum number of processors required to execute the static schedule produced by our algorithms. Finally, we demonstrate our methods on a specific example.

## Categories and Subject Descriptors

J.6 [Computer-Aided Engineering]: Computer-Aided Design (CAD)

## General Terms

Algorithms, Design, Performance, Theory

## 1. INTRODUCTION

Because the most time-critical parts of real-time or computation-intensive applications are loops, we must explore the parallelism embedded in the repetitive pattern of a loop. A loop can be modeled as a *data-flow graph* (DFG). The nodes of a DFG represent tasks, while edges between nodes represent data dependencies among tasks. Each edge may contain a number of delays (i.e. loop-carried dependencies). This model is widely used in many fields.

In our previous work [6–9], we proposed an efficient algorithm, *extended retiming*, which transforms a DFG into an equivalent graph with maximum parallelism. Indeed, we have demonstrated that extended retiming, when combined with minimum unfolding, achieves rate optimality, the first method we are aware of that this can be said about. However, the result of extended retiming is a graph containing split nodes. Indeed, it is a particular type of split-node graph which can be scheduled rate optimally. This is not to say

that we are physically altering the DFG by placing registers inside of functional units. Rather, we are describing an abstraction for a graph which provides a feasible schedule with loop pipelining. While the split-node graph is the most compact means for expressing this schedule, the properties of such graphs and the means by which they can be manipulated and this pipelined schedule drawn out have not been explored in the literature. We take the first step in this exploration in this paper, exhibiting one method for scheduling the system represented by a split-node graph and optimizing it to execute on a minimal number of processors.

In [6] we demonstrated the effectiveness of our extended retiming transformation via several experiments. In all cases, we were able to achieve better results by using extended retiming, getting an optimal clock period while requiring less unfolding. The usefulness of this new transformation is clear, but interpreting the split-node graph that results from its application still presents a problem. Many scheduling algorithms for standard data-flow graphs exist throughout the literature [1, 2]. There are even many techniques for reducing such schedules so that they require a minimal number of processors [3–5]. The problem is not new but the model is. In using a split-node DFG to represent a situation, we are conveying not only that a schedule is to be pipelined, but we are giving specific clues as to *how* it is to be pipelined. Even after we produce a schedule which obeys the additional rules regarding pipelining that the split-node graph dictates, most of the existing scheduling methods assume unlimited available processors, an unrealistic situation. We not only demonstrate a scheduling method for our new model, we then optimize the results of our method to require minimal hardware.

In this paper, we formally define a split-node data-flow graph and redefine the terminology of scheduling to fit this new paradigm. We demonstrate a scheduling algorithm for split-node graphs, and then discuss a way to reduce the hardware requirements of the resulting schedule. In the process, we state and prove a tight upper bound on the minimum number of processors required to execute our static schedule. Finally, we demonstrate our methods on a specific example.

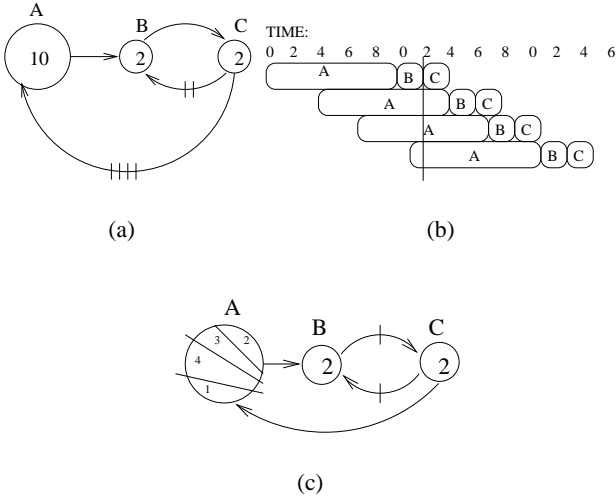
## 2. BACKGROUND

Before proceeding to our primary results, we first introduce our basic models. We then review previously established results pertinent to our task.

In our previous work [6, 8], we defined *extended retiming*, a graph transformation technique which minimizes the iteration period of a DFG by redistributing delays among the edges and inside of the nodes. One method for constructing the extended retiming function based on a DFG's static schedule was proposed in [7, 9]. We begin with the schedule defined in [1], based on the DFG's

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

GLSVLSI'02, April 18-19, 2002, New York, New York, USA.  
Copyright 2002 ACM 1-58113-462-2/02/0004 ...\$5.00.



**Figure 1: (a) A sample DFG; (b) The DFG schedule for Figure 1(a); (c) Figure 1(a) retimed.**

scheduling graph. (We will refer to this algorithm as *DFG scheduling* and will describe it in more detail later.) We now find the last node of the first iteration to be scheduled and cut the schedule at this point. We then read the retiming immediately by counting the occurrences of the nodes to the left of the cut.

For example, consider the DFG of Figure 1(a). Adopting a clock period of 7 and unfolding factor of 2, followed by the application of DFG scheduling, results in the schedule of Figure 1(b).  $C$  is the last node of the first iteration to be scheduled at time step 12, so we cut our diagram at this point as shown. To the left of the cut we see one complete copy of  $A$ , plus partial copies having times 1, 5 and 8. When we now retime node  $A$ , we pass one delay entirely through the node, while the remaining delays get stuck at these designated positions within  $A$ . The result is the split-node graph of Figure 1(c).

An *iteration* is simply an execution of all nodes in a data-flow graph (DFG) once. The average computation time of an iteration is called the *iteration period* of the DFG. If a DFG  $G$  contains a loop, then this iteration period is bounded from below by the *iteration bound* [10] of  $G$ , which is denoted  $B(G)$  and is the maximum time-to-delay ratio of all cycles in  $G$ . For example, there are two loops in Figure 1(a): the outer  $A \rightarrow B \rightarrow C \rightarrow A$  loop with total computation time 14 and delay count 4; and the  $B \rightarrow C \rightarrow B$  loop with time 4 and delay count 2. The larger of these ratios comes from the outer loop, and so  $B(G) = \frac{7}{2}$  in this case. In fact, the schedule of Figure 1(b) achieves this minimal iteration period, with 4 iterations being scheduled every 14 time steps. When the iteration period of the schedule equals the iteration bound of the DFG (as happens here), we say that the schedule is *rate-optimal*. Clearly, if we have a legal schedule for  $G$  with cycle period  $c$  and unfolding factor  $f$ , its iteration period is  $\frac{c}{f}$ , and since  $B(G)$  is a lower bound for the iteration period, we must have  $B(G) \leq \frac{c}{f}$ .

We can now define a *split-node data-flow graph (SDG)* with splitting degree  $\delta$  to be a finite, directed, weighted graph  $G = \langle V, E, d, t \rangle$  where:

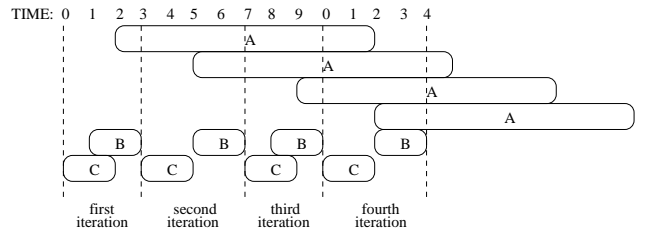
1.  $V$  is a vertex set;
2.  $E \subseteq V \times V$  is the edge set, representing precedence relations

among the nodes;

3.  $d : E \rightarrow \mathbf{Z}$  is a function with  $d(e)$  the delay count for edge  $e$ ;
4.  $t : V \rightarrow \mathbf{Z}^\delta$  is a function with the  $\delta$ -tuple  $t(v)$  representing the computation times of  $v$ 's pieces.

Broadly speaking,  $\delta$  is the maximum number of pieces any node of  $G$  is split into. If a node  $v$  is not split  $t(v)$  is an integer rather than a  $\delta$ -tuple. For example, in the SDG of Figure 1(c),  $t(A) = (1, 4, 3, 2)$  while  $t(B) = t(C) = 2$ . We will use the notation  $T(u)$  to refer to the sum of the elements of  $u$ 's  $\delta$ -tuple if  $u$  is split and to  $t(u)$  otherwise. In this example,  $T(A) = 10$  and  $T(B) = T(C) = 2$ .

In our model, delays may be contained either along an edge or within a node. Delays contained along an edge represent precedence relations across iterations; for example, the one-delay edge between  $B$  and  $C$  in Figure 1(c) indicates that the execution of  $B$  in the current iteration must terminate before  $C$  can begin in the next iteration. On the other hand, delays within a node convey information regarding the pipelined execution of a node. For example, the three delays inside of  $A$  tell us that up to 4 copies of the node may be executing simultaneously in a pipelined schedule of tasks. Furthermore, the position of the delays inside of a node indicate the form of the schedule of tasks for a graph. For example, one schedule for the SDG of Figure 1(c) appears in Figure 2. As we can see, the first iteration contains the beginning of  $A$ 's first copy; the next iteration includes only the part of this copy taking 4 time units to execute; the next iteration lasts only 3 time units to match the next part of the copy; and the next iteration includes the remaining piece of  $A$ 's first copy. After that, we schedule the copies of  $B$  and  $C$  as best we can around the borders of the iterations, making sure that the copy of  $B$  in this iteration precedes the copy of  $C$  in the next iteration, and that the current copy of  $A$  starts upon termination of the current copy of  $C$ .



**Figure 2: One schedule for Figure 1(c).**

Given an edge  $e = (u, v)$  in a SDG  $G$ , we will use the traditional notation  $d(e)$  to refer to the number of delays on the edge not including delays within end nodes. We will use  $d(u \rightarrow v)$  to denote the total number of delays along an edge, including delays contained within the end nodes. We will further define  $d^+(u \rightarrow v)$  as  $d(e)$  plus the number of delays within the source node  $u$ , and  $d^-(u \rightarrow v)$  as  $d(e)$  plus the number of delays within the sink node  $v$ . Referring to Figure 1(c), we observe that  $d^+(A \rightarrow B) = d(A \rightarrow B) = 3$  while  $d^-(A \rightarrow B) = d(e) = 0$  for  $e = (A, B)$ . It is easy to see that  $d(u \rightarrow v) = d^+(u \rightarrow v) + d^-(u \rightarrow v) - d(e)$  for any edge  $e = (u, v)$ .

An *integral time schedule* or *integral schedule* is a function  $s : V \times \mathbf{N} \rightarrow \mathbf{Z}$  where the starting time of node  $v$  in the  $i^{\text{th}}$  iteration is given by  $s(v, i)$ . It is a *legal schedule* if  $s(u, i) + T(u) \leq s(v, i + d^+(u \rightarrow v))$  for all edges  $e = (u, v)$  and iterations  $i$ , while a legal schedule is a *repeating schedule for cycle period  $c$*

and unfolding factor  $f$  if  $s(v, i + f) = s(v, i) + c$  for all nodes  $v$  and iterations  $i$ . Such a schedule can be represented by its first  $f$  iterations, since a new occurrence of this partial schedule can be started at the beginning of every interval of  $c$  clock ticks to form the complete legal schedule.

Given a DFG  $G$  without split nodes, a clock period  $c$  and an unfolding factor  $f$ , we construct the *scheduling graph*  $G^s = \langle V, E, w, t \rangle$  by reweighting each edge  $e = (u, v)$  according to the formula  $w(e) = d(e) - \frac{f}{c} \cdot t(u)$ . We then further alter  $G^s$  by adding a node  $v_0$  and zero-weight directed edges from  $v_0$  to every other node in  $G$ . We then let  $sh(v)$  be the length of the shortest path from  $v_0$  to  $v$  in  $G^s$  for every node  $v$ . It was demonstrated in [1] that, if  $B(G) \leq \frac{c}{f}$  for a given cycle period  $c$  and unfolding factor  $f$ , then the function  $S(v, i) = \left\lceil \frac{c}{f}(i - sh(v)) \right\rceil$  for all nodes  $v$  and positive integers  $i$  is a legal, integral, repeating schedule. In fact, this is the method used to construct the schedule in Figure 1(b).

Given a set of processors  $L$ , a *processor assignment* or *assignment* is a function  $p : V \times \mathbf{N} \rightarrow L$  where node  $v$  in iteration  $i$  is executed using processor  $p(v, i)$ . An assignment with unfolding factor  $f$  is *static* (with its corresponding time schedule a *static schedule*) if  $p(v, i + f) = p(v, i)$  for every iteration  $i$ . Finally, we can implement a static schedule using one of two design styles. If two copies of a node cannot be in execution simultaneously, the schedule follows a *non-pipelined implementation*. This creates an implicit precedence relation between consecutive copies of the same node, insuring that the first copy stops before the succeeding copy begins. If no such restriction is placed on the schedule, it is said to follow a *pipelined implementation*. As eluded to earlier, we will assume the use of pipelining throughout this paper.

### 3. RESOURCE MINIMIZATION

Having outlined a method by which the nodes of a SDG may be scheduled, we now explore the problem of assigning them to processors for execution. The simplest method is to statically assign *dedicated processors*, where a functional unit executes iterations of one and only one node from the schedule. Due to pipelining, it is necessary to assign multiple processors to a single node in order to handle overlapping iterations. We may then ask exactly how many processors are required under these rules:

**THEOREM 3.1.** *Let  $G = \langle V, E, d, t \rangle$  be a data-flow graph. Given the static repeating integral schedule  $S$  from above having clock period  $c$  and unfolding factor  $f$ , execution of all iterations of node  $v$  may be completed using  $\left\lceil \frac{f}{c} \cdot T(v) \right\rceil$  dedicated processors for each  $v \in V$ .*

**PROOF.** Iterations  $i$  and  $i + k$  of node  $v$  can share one processor if and only if  $S(v, i) + T(v) \leq S(v, i + k)$ , which by definition occurs if and only if

$$\left\lceil \frac{c}{f}(i - sh(v)) \right\rceil + T(v) \leq \left\lceil \frac{c}{f}(i + k - sh(v)) \right\rceil.$$

Since  $T(v)$  is integral, this is equivalent to  $\left\lceil \frac{c}{f}(i - sh(v)) + \frac{ck}{f} \right\rceil - \left\lceil \frac{c}{f}(i - sh(v)) + T(v) \right\rceil \geq 0$ , which happens if and only if  $\frac{ck}{f} - T(v) > -1$  and  $k > \frac{f}{c}(T(v) - 1)$ .

We now have two cases. First suppose that  $\frac{f}{c}(T(v) - 1)$  is integral. Clearly  $\frac{f}{c}(T(v) - 1) < \frac{f}{c} \cdot T(v) \leq \left\lceil \frac{f}{c} \cdot T(v) \right\rceil$ . On the other hand,  $\left\lceil \frac{f}{c} \cdot T(v) \right\rceil = \left\lceil \frac{f}{c}(T(v) - 1) + \frac{f}{c} \right\rceil \leq \frac{f}{c}(T(v) - 1) + 1$  since

$0 < f < c$  by assumption. Therefore  $\left\lceil \frac{f}{c} \cdot T(v) \right\rceil$  is the smallest integer strictly greater than  $\frac{f}{c}(T(v) - 1)$  and so is the minimum number of processors required to execute all iterations of  $v$ . Otherwise  $\frac{f}{c}(T(v) - 1)$  is not integral and we need at least  $\left\lceil \frac{f}{c}(T(v) - 1) \right\rceil$  processors, which is also bounded above by  $\left\lceil \frac{f}{c} \cdot T(v) \right\rceil$ .

In any case, if  $n = \left\lceil \frac{f}{c} \cdot T(v) \right\rceil$ , we have demonstrated above that one processor is adequate for executing  $v_0, v_n, v_{2n}$ , and so on; the same processor can be assigned to  $v_1, v_{n+1}, v_{2n+1}$ , etc.;...; and one processor for  $v_{n-1}, v_{2n-1}, v_{3n-1}$ , and so forth. Thus all iterations of node  $v$  may be executed using only  $n$  processors.  $\square$

For instance, consider the example of Figure 1(c) again with clock period 7 and unfolding factor 2. By this result, we require  $\left\lceil \frac{20}{7} \right\rceil = 3$  dedicated processors for  $A$  and  $\left\lceil \frac{4}{7} \right\rceil = 1$  dedicated processor for each of  $B$  and  $C$ . In fact, our schedule for Figure 1(c) reserved 5 processors for execution, one for each ‘‘row’’ of the schedule. However, this may be an overestimate. For example, in the schedule of Figure 2, there is no compelling reason why some iterations of  $B$  and  $C$  cannot share a processor, thus possibly reducing the hardware cost needed to implement our final schedule. We are now ready to explore this question of systematically studying a given static schedule in an attempt to produce a processor assignment using undedicated processors and requiring minimal hardware.

Our processor assignment method is based on the ideas from [3], although we improve on their idea by considering rational as well as integral iteration periods and fill certain logical gaps. Given a clock period  $c$ , we first divide time into *segments*, each containing  $c$  clock ticks. Segment 1 lasts from time step 0 until time step  $c$ , segment 1 from  $c$  to  $2c$ , and in general segment  $k$  lasts from time step  $(k-1) \cdot c$  until time step  $k \cdot c$ . We then *unfold* nodes, dividing the first  $f$  copies of each node in our schedule so that the pieces are small enough to fit into one segment, where  $f$  is our given unfolding factor. We use an unfolding method similar to that of [4, 5], simplifying their calculations by assuming that all nodes begin execution at least once during the first segment of our schedule. In general we have two cases.

1. If a node  $v$  is small enough to begin and complete execution of its zeroth iteration during the first segment of the schedule, it remains in one piece. We simply pass its starting and computation times to the next stage of our method as  $\sigma(v)$  and  $\tau(v)$ , respectively. We also mark it as having no preceding piece ( $\rho(v) = \text{NIL}$ ).
2. Otherwise the node overlaps segments and must be divided into parts. The first part, the *head*, is assigned at the end of the segment and includes the first part of the node which executes until the end of the first segment. (It thus inherits its starting time from the original node and has a computation time of the clock period minus the start time.) Next are the *body* sections which span entire subsequent segments. Finally, the *tail* is assigned at the beginning of a segment and completes execution of the original node following all body sections. In all cases, we assign  $u(u)$  to identify what type of node piece  $u$  is. We also assign the identity of the preceding piece to  $\rho(u)$ .

Since the pieces we are creating must refer back to their matching node from the original schedule, we use  $org(u)$  and  $copy(u)$  to identify the source and iteration, respectively, in the case of HEAD or whole nodes. This entire procedure appears as Algorithm 1.

As an example, consider the schedule from Figure 2. The first two iterations of both  $B$  and  $C$  begin and complete execution within the first 7 time steps, so are passed on to the next stage of our

---

**Algorithm 1** Folding the vertices of a split-node graph
 

---

**Input:** A SDG  $G = \langle V, E, d, t \rangle$ , a schedule  $S$  for  $G$  having clock

period  $c$  and unfolding factor  $f$

**Output:** A set of folded vertices  $W$

```

 $W \leftarrow \emptyset$  /*  $W$  is the set of folded nodes */
for all  $v \in V$  do
  for  $i \leftarrow 0$  to  $f - 1$  do
     $m \leftarrow \left\lceil \frac{T(v) + S(v, i)}{c} \right\rceil - \left\lfloor \frac{S(v, i)}{c} \right\rfloor$ 
     $W \leftarrow W \cup \{v_{i0}, v_{i1}, \dots, v_{i(m-1)}\}$ 
  if  $m = 1$  then
     $\tau(v_{i0}) \leftarrow T(v)$  /* Computation time */
     $\sigma(v_{i0}) \leftarrow S(v, i)$  /* Start time */
     $\rho(v_{i0}) \leftarrow \text{NIL}$  /* Preceding piece of folded node */
     $\iota(v_{i0}) \leftarrow \text{NIL}$  /* Identification; head, body, tail or none? */
     $org(v_{i0}) \leftarrow v$  /* Corresponding original node from  $V$  */
     $copy(v_{i0}) \leftarrow i$  /* Iteration of original node */
  else
     $\tau(v_{i0}) \leftarrow c - S(v, i)$ 
     $\sigma(v_{i0}) \leftarrow S(v, i)$ 
     $\rho(v_{i0}) \leftarrow \text{NIL}$ 
     $\iota(v_{i0}) \leftarrow \text{HEAD}$ 
     $org(v_{i0}) \leftarrow v$ 
    for  $j \leftarrow 1$  to  $m - 2$  do
       $\tau(v_{ij}) \leftarrow c$ 
       $\sigma(v_{ij}) \leftarrow 0$ 
       $\rho(v_{ij}) \leftarrow v_{i(j-1)}$ 
       $\iota(v_{ij}) \leftarrow \text{BODY}$ 
       $org(v_{ij}) \leftarrow \text{NIL}$ 
       $copy(v_{ij}) \leftarrow \text{NIL}$ 
    end for
     $\tau(v_{i(m-1)}) \leftarrow T(v) - \tau(v_{i0}) - c \cdot (m - 2)$ 
     $\sigma(v_{i(m-1)}) \leftarrow 0$ 
     $\rho(v_{i(m-1)}) \leftarrow v_{i(m-2)}$ 
     $\iota(v_{i(m-1)}) \leftarrow \text{TAIL}$ 
     $org(v_{i(m-1)}) \leftarrow \text{NIL}$ 
     $copy(v_{i(m-1)}) \leftarrow \text{NIL}$ 
  end if
end for
end for

```

---

method as they are. On the other hand, the execution of  $A_0$  spans the first two segments, so must be divided into 2 pieces. The HEAD section executes from time step 2 until time step 7 and the TAIL from 7 to 12. Similarly, the execution of  $A_1$  is divided into 3 pieces, with the HEAD section executing from time step 5 until time step 7, the BODY section from 7 to 14, and the TAIL from 14 to 15. The results of applying Algorithm 1 to these nodes, as well as a summary of their properties from the original schedule, are given in Table 1.

We next identify pieces that can be scheduled back-to-back on the same processor by creating *independent lists* of folded node pieces. We first sort our nodes, using starting time as the first key and “subscript” as the second. (In other words, if starting times are all equal, the HEAD of a node has higher priority than any of the BODY pieces, all of which have higher priority than the TAIL.) If a node drawn from this sorted list has starting time zero, a new

$v$	$T(v)$	$i$	$S(v, i)$	$m$		$\sigma$	$\tau$	$\rho$	$\iota$	$org$	$copy$	
A	10	0	2	2		$A_{00}$	2	5	NIL	HEAD	A	0
						$A_{01}$	0	5	$A_{00}$	TAIL	NIL	NIL
		1	5	3		$A_{10}$	5	2	NIL	HEAD	A	1
						$A_{11}$	0	7	$A_{10}$	BODY	NIL	NIL
B	2	0	1	1		$B_{00}$	1	2	NIL	NIL	B	0
						$B_{10}$	5	2	NIL	NIL	B	1
		1	5	1		$C_{00}$	0	2	NIL	NIL	C	0
						$C_{10}$	3	2	NIL	NIL	C	1

**Table 1: Results from Algorithm 1 for the schedule of Figure 2.**

---

**Algorithm 2** Create independent lists from folded nodes
 

---

**Input:** A set of folded nodes  $W$  and clock period  $c$

**Output:** Independent lists of folded vertices

```

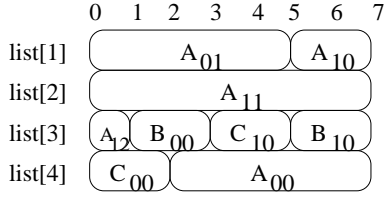
 $Q \leftarrow \emptyset$  /* Priority queue of vertices, ordered by  $\sigma$  values first, subscript second. */
for all  $v \in W$  do
   $\varsigma(v) \leftarrow \text{NIL}$ 
  enqueue( $v, Q$ )
end for
/* Stack of indep. proc. lists unassigned from time step  $i$  forward */
for  $i \leftarrow 1$  to  $c$  do
  avail $[i] \leftarrow \emptyset$ 
end for
 $\ell \leftarrow 0$  /* number of indep. lists */
for  $i \leftarrow 1$  to  $|W|$  do
   $u \leftarrow \text{dequeue}(Q)$ 
  if  $\sigma(u) = 0$  then
     $\ell \leftarrow \ell + 1$ 
    list $[\ell] \leftarrow u$ 
    if  $\rho(u) \neq \text{NIL}$  and  $\varsigma(\rho(u)) = \text{NIL}$  then
       $\varsigma(\rho(u)) \leftarrow \ell$ 
    end if
    push list $[\ell]$  onto avail $[\tau(u)]$ 
  else
    if avail $[\sigma(u)] \neq \emptyset$  then
      pop list $[j]$  from avail $[\sigma(u)]$  /* if indep. list can hold  $u$  add it to list */
      list $[j] \leftarrow \text{list}[j] \cup \{u\}$ 
      push list $[j]$  onto avail $[\sigma(u) + \tau(u)]$ 
    else
       $\ell \leftarrow \ell + 1$  /* otherwise need a new list */
      list $[\ell] \leftarrow u$ 
      push list $[\ell]$  onto avail $[\sigma(u) + \tau(u)]$ 
    end if
  end if
end for
for  $i \leftarrow 1$  to  $\ell$  do
  for all  $u \in \text{list}[i]$  do
    if  $\sigma(u) + \tau(u) = c$  then
      succ $(i) \leftarrow \varsigma(u)$ 
    end if
    head $(i) \leftarrow \text{head}(i)$  or  $\iota(u) = \text{HEAD}$ 
    body $(i) \leftarrow \text{body}(i)$  or  $\iota(u) = \text{BODY}$ 
    tail $(i) \leftarrow \text{tail}(i)$  or  $\iota(u) = \text{TAIL}$ 
  end for
end for
end for

```

---

list is created which contains only this node. In this case, we also mark its preceding piece with the number of the current list so that lists containing pieces from the same original node can be placed in proper order later. (In our algorithm, this step is accomplished by assigning a  $\varsigma$  value to any node which is discovered to be the  $\rho$  value of a piece we are assigning to a list.) On the other hand, if a node from the list has a non-zero starting time, we examine all of our previous lists to see if one is unoccupied at the given starting time. If such a list is found, we add the current node to it. Otherwise, we must start a new list to accommodate this node. At the end, we examine all independent lists. If the last node added to a list has a non-NIL  $\varsigma$  value, we set the *succ* value of the list equal to this, thus identifying the list which must follow the current one when assigning lists to a processor. We also record whether the current list contains HEAD, BODY or TAIL pieces for use in the next step of our method. All of this is accomplished by Algorithm 2 below.

We resume our example with the data in Table 1, which yields a priority queue of  $\{A_{01}, A_{11}, A_{12}, C_{00}, B_{00}, A_{00}, C_{10}, A_{10}, B_{10}\}$ . The first four items are assigned to *list*[1] through *list*[4], respectively, with notes made that *list*[1] is available past time step 5, *list*[3] past step 1 and *list*[4] past step 2. In the process, we discover three of the nodes ( $A_{01}$ ,  $A_{11}$  and  $A_{12}$ ) with non-NIL  $\rho$  values, so must set the appropriate  $\varsigma$  values. In this case,  $\varsigma(A_{00}) = 1$  since  $\rho(A_{01}) = A_{00}$  and  $A_{01}$  was added to *list*[1], while  $\varsigma(A_{10}) = 2$  and  $\varsigma(A_{11}) = 3$  for similar reasons. When we pop  $B_{00}$  from the queue and discover it has starting time 1, we add it to *list*[3] rather than begin a new list because of our note regarding that list. We continue in this fashion, eventually arriving at the list assignments pictured in Figure 3(a). We next review the four lists and assign non-NIL *succ* values to *list*[1], *list*[2] and *list*[4] since these lists contain the nodes with non-NIL  $\varsigma$  values. Finally, we note whether any list contains HEAD, BODY or TAIL nodes. These values that we have derived are summarized in Figure 3(b).



(a)

	<i>succ</i>	<i>head</i>	<i>body</i>	<i>tail</i>
<i>list</i> [1]	2	TRUE	FALSE	TRUE
<i>list</i> [2]	3	FALSE	TRUE	FALSE
<i>list</i> [3]	NIL	FALSE	FALSE	TRUE
<i>list</i> [4]	1	TRUE	FALSE	FALSE

(b)

**Figure 3: Results of Algorithm 2 applied to the data from Table 1: (a) the independent lists; (b) the lists' data values**

We next take our independent lists and use them to make an initial processor assignment. We first split the lists into two groups, those which contain BODY or TAIL pieces and those that do not. If possible, we select a list without such pieces and assign it to the first available processor. If this list contains a HEAD or BODY node, we know that there is another list which must be assigned to the same processor immediately following our current list, so we find this new list and add it to the current group. Simultaneously, we increment our unfolding factor to account for pipelining. We have assumed that the zeroth iteration of all nodes begins execution during the first clock segment. Therefore, if our first contact with a node in this part of our procedure takes place during a subsequent clock segment, we know that the node represents a later iteration. This incremented unfolding factor helps us keep track of which copy of a node we are actually placing. Finally, after completing a group of lists, we increment the processor count by the unfolding factor to account for pipelining, so that the next group may be assigned to the correct new processor. This procedure appears as Algorithm 3 below.

For example, apply this method to the data from Figures 3(a) and 3(b). Our queue of lists without BODY or TAIL sections is  $\{4\}$ , while all other lists are in the other queue. With no other choice, we begin by assigning *list*[4] to the first processor. Thus the zeroth iterations of  $C_0$  and  $A_0$  are scheduled to execute here. The *succ* of this list is 1, so we remove this list from our other queue, increment the unfolding factor to 1, and schedule the new list's execution on processor one during the second time segment. This list contains the HEAD of  $A_1$ , so we schedule the first iteration (due to the unfolding factor) of this node to execute on our processor. We proceed as before, incrementing the unfolding factor (to 2) and loading the *successor* to *list*[1], namely *list*[2]. Since this new list contains only the BODY of  $A_1$  no values are altered. Finally, we increment the unfolding factor and process this list's successor, *list*[3]. This new list contains the only copies of  $B_0$ ,  $C_1$  and  $B_1$ , so these nodes are scheduled for processor one during their third iteration (which correspond to nodes  $B_6$ ,  $C_7$  and  $B_7$ , respectively, of the original

### Algorithm 3 Use independent lists to assign processors

**Input:** Set of  $\ell$  independent lists

**Output:** A processor assignment for iteration zero

```

Q ← ∅ /* Sorted list of unproc. lists w/o body or tail */
R ← ∅ /* Sorted list of all other unproc. lists */
for i ← 1 to  $\ell$  do
  proc(i) ← FALSE /* Set all lists unproc. */
  if body(i) = FALSE and tail(i) = FALSE then
    Q ← Q ∪ {i}
  else
    R ← R ∪ {i}
end for
p ← 0 /* Number of processors */
while TRUE do
  if Q ≠ ∅ or R ≠ ∅ then
    p ← p + 1 /* If ∃ unproc. list increment p */
  else
    stop /* Otherwise halt */
  end if
  if Q ≠ ∅ then
    l ← first element of Q /* Select list w/o body or tail if you can */
  else
    l ← first element of R
  end if
  current ← ∅ /* current schedule piece and its unfolding factor */
  f ← 0
  while proc(l) = FALSE do
    proc(l) ← TRUE
    current ← current ∪ list[l]
    for all u ∈ list[l] do
      if  $\iota(u)$  = NIL or  $\iota(u)$  = HEAD then
        unit(u) ← p /* Assign proc. p to node u iter. f */
        iter(u) ← f
      end if
    end for
    if head(l) = TRUE or body(l) = TRUE then
      t ← l /* Select list containing body or tail following current list */
      l ← succ(t)
      if body(l) = TRUE or tail(l) = TRUE then
        if l ∈ R then remove l from R end if
      else
        if l ∈ Q then remove l from Q end if
      end if
    end if
    f ← f + 1 /* For each new appended list increment unfolding factor */
  end while
  for all u ∈ current do
    if  $\iota(u)$  = NIL or  $\iota(u)$  = HEAD then
      unfold(u) ← f /* Assign unlf. factor to all scheduled nodes */
    end if
  end for
  p ← p + f
end while

```

schedule). The queues are now both empty and every node in our graph has been scheduled to execute on the same processor at varying points in time. We finish by assigning all nodes an *unfold* value of 3, leaving us with the values for each node listed in Table 2.

Finally, we must propagate this initial processor assignment throughout the remaining iterations of our nodes. In Algorithm 3, we assigned a processor for a designated iteration of each node. We also recorded an unfolding factor, telling us that  $unfold(u) + 1$  copies of this particular node  $u$  are in the pipeline simultaneously. It is now a simple task to assign the  $unfold(u) + 1$  copies of node  $u$  to the processors we reserved for these tasks, then repeat this assignment statically to infinity, thus completing our processor assignment. This final step is summarized in Algorithm 4.

	<i>unit</i>	<i>iter</i>	<i>unfold</i>
$A_{00}$	1	0	3
$A_{10}$	1	1	3
$B_{00}$	1	3	3
$B_{10}$	1	3	3
$C_{00}$	1	0	3
$C_{10}$	1	3	3

**Table 2: The output from applying Algorithm 3 to the data in Figures 3(a) and 3(b).**

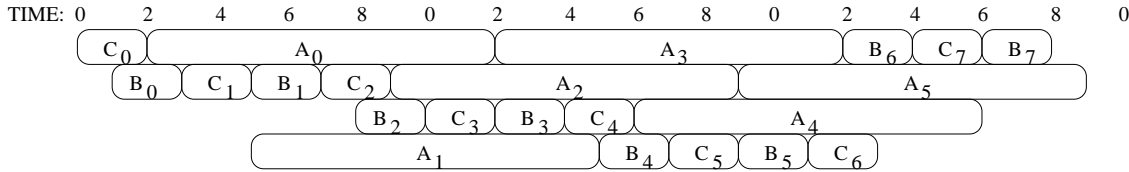


Figure 4: Final processor-optimal schedule for example.

---

**Algorithm 4** Expand iter. zero proc. assignment to complete proc. assignment

---

**Input:** Set of folded nodes  $W$  with assigned values and unfolding factor  $f$

**Output:** A complete processor assignment

```

for all  $u \in W$  with  $\iota(u) \neq \text{NIL}$  or  $\iota(u) \neq \text{HEAD}$  do
  for  $i \leftarrow 0$  to  $f - 1$  do
     $k \leftarrow f \cdot i + \text{copy}(u)$ 
    for  $j \leftarrow 0$  to  $\text{unfold}(u)$  do
       $P(\text{org}(u), k + ((\text{iter}(u) + j) \bmod (\text{unfold}(u) + 1)))$ 
       $\leftarrow \text{unit}(u) + j$ 
    end for
  end for
  /* Repeat first few iters. to infinity */
  for all  $i \geq 0$  do
     $P(\text{org}(u), i + f \cdot (\text{unfold}(u) + 1)) \leftarrow P(\text{org}(u), i)$ 
  end for
end for

```

---

We now complete our example, using the values from Figure 2. Due to our unfolding factors, we require 4 processors to complete our schedule. We see from Table 2 that the first iteration of  $A_1$  (i.e.,  $A_3$ ) is assigned to processor 1. Working backward, this means that the initial iteration of the node ( $A_1$ ) is placed on processor 4. Also, the second iteration  $A_5$  is assigned to processor 2, the third to processor 3, the fourth to processor 4, and so on. Similarly, if the third iteration of  $B_0$  (really  $B_6$ ) is assigned to the first processor, we work backward to assign the second iteration ( $B_4$ ) to processor 4, the first iteration ( $B_2$ ) to processor 3, and the zeroth iteration to processor 2. We do the exact same thing to assign the iterations of  $B_1$  and  $C_1$ . Finally, the remaining nodes all have their zeroth iterations scheduled for processor 1, so it is easy to work forward and complete our processor assignment. We thus derive the final time- and processor-optimal schedule seen in Figure 4.

## 4. CONCLUSION

In this paper, we have formally defined a split-node data-flow graph and redefined the terminology of scheduling to fit this new paradigm. We have exhibited one scheduling algorithm for split-node graphs, and then discussed a way to reduce the hardware requirements of the resulting schedule. In the process, we stated and proved a tight upper bound on the minimum number of processors required to execute the static schedule produced by our algorithms. Finally, we have demonstrated our methods on a specific example.

## 5. ACKNOWLEDGEMENTS

This work is partially supported by NSF grants MIP-95-01006 and MIP-97-04276, and by the A. J. Schmitt Foundation.

## 6. REFERENCES

- [1] L.-F. Chao and E. H.-M. Sha. Static scheduling for synthesis of DSP algorithms on various models. *Journal of VLSI Signal Processing*, 10:207–223, 1995.
- [2] G. DeMicheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, Inc., 1994.
- [3] Y.-C. Ho and J.-C. Tsay. Fully static processor-optimal assignment of data-flow graphs. *IEEE Signal Processing Letters*, 4:146–148, 1997.
- [4] K. Ito, L. Lucke, and K. Parhi. Module selection and data format conversion for cost-optimal DSP synthesis. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 322–329, 1994.
- [5] L. Lucke and K. Parhi. Generalized ILP scheduling and allocation for high-level DSP synthesis. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 5.4.1–5.4.4, 1993.
- [6] T. O’Neil and E. H.-M. Sha. Rate-optimal graph transformation via extended retiming and unfolding. In *Proceedings of the IASTED 11th International Conference on Parallel and Distributed Computing and Systems*, volume 10, pages 764–769, 1999.
- [7] T. O’Neil and E. H.-M. Sha. Optimal graph transformation using extended retiming with minimal unfolding. In *Proceedings of the IASTED 12th International Conference on Parallel and Distributed Computing and Systems*, volume I, pages 128–133, 2000.
- [8] T. O’Neil, S. Tongsima, and E. H.-M. Sha. Extended retiming: Optimal retiming via a graph-theoretical approach. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 4, pages 2001–2004, 1999.
- [9] T. O’Neil, S. Tongsima, and E. H.-M. Sha. Optimal scheduling of data-flow graphs using extended retiming. In *Proceedings of the ISCA 12th International Conference on Parallel and Distributed Computing Systems*, pages 292–297, 1999.
- [10] M. Renfors and Y. Neuvo. The maximum sampling rate of digital filters under hardware speed. *Transactions on Circuits and Sampling*, CAS-28:196–202, 1981.