Time-Constrained Loop Scheduling with Minimal Resources

Timothy W. O'Neil*
Dept. of Comp. Science
University of Akron
Akron, OH 44325-4003
Phone: (330) 972-6492
Fax: (330) 374-8630
Email: toneil@cs.uakron.edu

Edwin H.-M. Sha
Dept. of Comp. Science
Erik Jonsson School of Eng. & C.S.
Box 830688, MS EC 31
Univ. of Texas at Dallas
Richardson, TX 75083-0688
Phone: (972) 883-4193
Fax: (972) 883-2349
Email: edsha@utdallas.edu

Abstract

Many 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 develop new results regarding the construction of such graphs. We develop scheduling algorithms for such graphs, and then discuss a way to reduce the hardware requirements of such schedules. 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. We also construct an unfolding algorithm for split-node graphs and combine it with our scheduling methods to achieve rate-optimality in all cases. Finally, we demonstrate our methods on a specific example.

This work was partially supported by NSF grants MIP95-01006 and MIP97-04276, and by the A. J. Schmitt Foundation while the authors were with the University of Notre Dame. It is currently supported by the University of Akron; and by the TI University Program, NSF ETA 0103709, Texas ARP 009741-0028-2001.

Keywords: Parallel and Distributed Systems, Compilers, Optimization, Modeling Languages, Programmable Signal Processors

I. 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*) [1]. 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, including circuitry [2], digital signal processing [3] and program descriptions [4].

In our previous work [5]–[8], 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 (hereafter referred to as a *split-node graph*). In fact, 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 perform this exploration herein.

In [6] we demonstrated the effectiveness of our extended retiming transformation via several experiments which are summarized in Table I. In all cases, we achieve better results by using extended retiming, getting an optimal clock period while requiring less unfolding. This improvement is illustrated by the last four columns of our table. Limiting ourselves to traditional retiming without node splitting forces us to decide between two poor options. If we want an optimal clock period while avoiding node splitting, we must unfold by a larger factor, which is listed for each example in the second-to-last column of Table I. This dramatically increases the size of our circuit, and thus the number of functional units we require and the production costs. On the other hand, if we want to unfold by our extended unfolding factor (shown in boldface in the table), we will be forced to accept a larger iteration period (listed in the last column of the same table) unless we split nodes. Otherwise the result is a smaller circuit running at less than optimal speed.

Benchmark	Computation Time		Slow-down	Iter. Bound	Min. Optimal Unf. Factor		Iter. Pd. w/ **Bold** Unf. Factor	
	Add	Mult			**Ext.**	Trad.	Ext.	Trad.
Second Order IIR Filter	1	4	2	3	**1**	2	3	4
Second Order IIR Filter	1	10	6	2	**1**	6	2	10
2-Cascaded Biquad Filter	4	25	6	5.5	**2**	6	5.5	12.5
All-Pole Lattice Filter	2	5	12	1.5	**2**	6	1.5	2.5
All-Pole Lattice Filter	1	12	7	4	**1**	7	4	12
Fifth Order Elliptic Filter	2	12	16	3.5	**2**	8	3.5	6
Fifth Order Elliptic Filter	2	30	20	5.5	**2**	20	5.5	15

TABLE I

EXPERIMENTAL RESULTS FOR COMMON CIRCUITS

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 [9]–[13]. There are even many techniques for reducing such schedules so that they require a minimal number of processors [14]–[16]. 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. While there is much that we can build on, we must make specific modifications to existing methods so that they apply to this new paradigm and produce a schedule which obeys the additional rules regarding pipelining that the split-node graph dictates. Furthermore, most of the existing methods assume unlimited available processors, an unrealistic situation. We not only develop a scheduling method for our new model, we then optimize the results of our method to require minimal hardware.

In addition, we can see from Table I that there are examples where extended retiming alone is not sufficient for complete optimization. As mentioned above, we show in [6] that rate optimality may always be achieved by combining extended retiming with *unfolding*, a graph transformation technique where multiple iterations of a DFG are represented simultaneously [17]. This implies that we must be able to unfold split-node DFGs, but as with scheduling we must first alter the existing unfolding algorithm to apply to our new model. We perform this modification in this paper.

In this paper, we formally define a split-node data-flow graph and redefine the terminology of scheduling to fit this new paradigm. We explore the properties of the particular idealized form of split-node graphs constructed by the methods of [7], [8]. We develop scheduling algorithms for split-node graphs, and then discuss a way to reduce the hardware requirements of such schedules. 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. We also construct an unfolding algorithm for split-node graphs and combine it with our scheduling methods to achieve rate-optimality in all cases. Finally, we demonstrate our methods on specific examples.

The next section features basic definitions, followed by preliminary results which aid in the understanding of how such graphs are constructed. We then outline scheduling algorithms which apply to this new model. Next, we develop techniques by which we can reduce the hardware requirements of these schedules. Following this, we develop our unfolding algorithm and demonstrate its effectiveness when combined with our other methods. We conclude with examples and point to future directions.

II. BACKGROUND

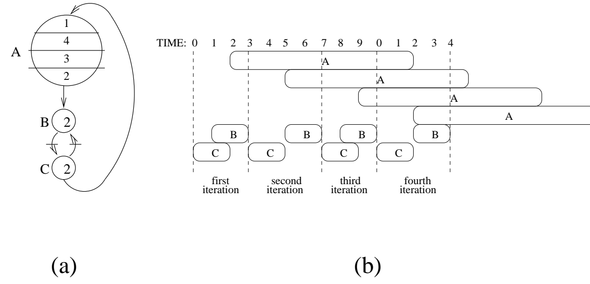
Before proceeding to our primary results, we first introduce our basic models. We then review previously established results pertinent to our task.

A *split-node data-flow graph* (*SDG*) with splitting degree δ is 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 δ-tuple $t(v)$ representing the computation times of v's pieces.

Broadly speaking, δ is the maximum number of pieces any node of G is split into by delays. (In fact, if $\delta = 1$ this becomes the traditional definition of a DFG.) By convention, if a node is split into n pieces where $n < \delta$, we will represent the computation time by an n-tuple rather than have null entires in a δ-tuple. Thus if a node v is not split $t(v)$ is an integer rather than a δ-tuple. For example, in the SDG of Figure 1(a), $t(A) = (1, 4, 3, 2)$ while $t(B) = t(C) = 2$. We will use the notation $T(u)$ to refer to the sum of the elements of u's δ-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. The execution of all nodes in V once is an *iteration*. Delays contained along an edge represent precedence relations across iterations; for example, the one-delay edge between B and C in Figure 1(a) 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



(a) (b)

Fig. 1. (a) A sample SDG; (b) One schedule for Figure 1(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(a) appears in Figure 1(b). 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.

The delay count of an edge in a SDG has been defined in depth in [21], [22]. For our purposes, 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 also define $d^+(u \rightarrow v)$ as $d(e)$ plus the number of delays within the source node u. Referring to Figure 1(a), we observe that $d^+(A \rightarrow B) = 3$.

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 ith 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.

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* [18] 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 $B(G) \leq \frac{c}{f}$.

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 [10] 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.

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.

III. SDGs Produced by Extended Retiming

In reality there is no restriction on how to split nodes in a DFG. However, we are interested in split-node graphs constructed by a particular method. In this section we will review this method and derive a couple of simple properties which pertain to the SDGs produced by our approach.

In our previous work [5], [6], 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], [8]. We begin with the schedule we defined above, based on the DFG's scheduling graph. (This algorithm, which we will refer to as *DFG scheduling*, first appeared in [10].) 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 2(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 2(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(a) which we have been working with from the start.

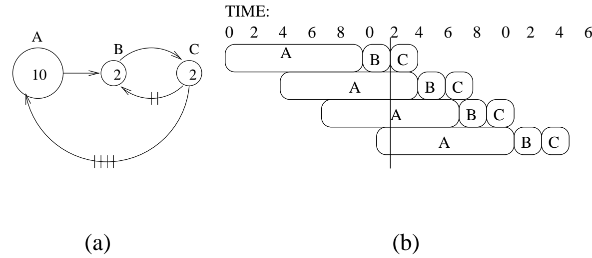


Fig. 2. (a) A sample DFG; (b) The DFG schedule for Figure 2(a).

Any node split via this process is split into subnodes which are either the *head*, the *tail* or one of many *internal nodes*. If v is a split node, we will use the functions $head(v)$ and $tail(v)$ to refer to the computation times of the head and tail, respectively. For example, node A of Figure 1(a) has $head(A) = 1$ and $tail(A) = 2$, with internal nodes having times 4 and 3. In general, we are interested in knowing what the computation times of these pieces are, leading us to this:

Theorem 3.1: Any internal piece of a node split by our method has computation time either $\left\lfloor \frac{c}{f} \right\rfloor$ or $\left\lceil \frac{c}{f} \right\rceil$.

Proof: In a SDG resulting from our method, the computation time of any internal piece of a split node is the difference in start times of consecutive iterations of that node. By definition $S(v,i) = \left\lceil \frac{c}{f}(i - sh(v)) \right\rceil$. Therefore $S(v,i+1) - S(v,i) = \left\lceil \frac{c}{f}(i+1-sh(v)) \right\rceil - \left\lceil \frac{c}{f}(i - sh(v) \right\rceil \leq \left\lceil \frac{c}{f} \right\rceil$ by the basic properties of the ceiling function. We now wish to demonstrate that $S(v,i+1) - S(v,i) \geq \left\lfloor \frac{c}{f} \right\rfloor$. There are two cases:

1) If $\frac{c}{f}(i - sh(v))$ is integral then $S(v,i) = \left\lfloor \frac{c}{f}(i - sh(v) \right\rfloor$. Clearly $S(v,i+1) \geq \left\lfloor \frac{c}{f}(i+1-sh(v)) \right\rfloor$ by definition and so $S(v,i+1) - S(v,i) \geq \left\lfloor \frac{c}{f}(i+1-sh(v)) \right\rfloor - \left\lfloor \frac{c}{f}(i - sh(v)) \right\rfloor \geq \left\lfloor \frac{c}{f} \right\rfloor$ by the basic properties of the floor function.
2) On the other hand assume that $\frac{c}{f}(i - sh(v))$ is not integral so that $S(v,i) = \left\lfloor \frac{c}{f}(i - sh(v) \right\rfloor + 1$. If

$\frac{c}{f}(i+1-sh(v))$ is also not integral then $S(v,i+1)-S(v,i) \geq \left\lfloor \frac{c}{f} \right\rfloor$ as above. Therefore assume that $\frac{c}{f}(i+1-sh(v))$ is integral. Then $S(v,i+1)-S(v,i) = \frac{c}{f}(i+1-sh(v)) - \left\lfloor \frac{c}{f}(i-sh(v) \right\rfloor - 1 = \left(\frac{c}{f}(i-sh(v)) - \left\lfloor \frac{c}{f}(i-sh(v)) \right\rfloor \right) + \frac{c}{f} - 1$. Since the quantity in parentheses is strictly between 0 and 1, $\frac{c}{f} - 1 < S(v,i+1) - S(v,i) < \frac{c}{f}$. Since $S(v,i+1)-S(v,i)$ must also be integral, we deduce that $S(v,i+1)-S(v,i) = \left\lfloor \frac{c}{f} \right\rfloor$ by the definition of the floor function.

In any case $\left\lfloor \frac{c}{f} \right\rfloor \leq S(v,i+1)-S(v,i) \leq \left\lceil \frac{c}{f} \right\rceil$ for any node v and integer $i \geq 0$. However, this quantity is integral and is sandwiched between consecutive integers. Therefore this quantity must equal one of these two bounds. ■

The computation times for a node's $head$ and $tail$ may be calculated according to the results from [19]. Knowing these values for each node is useful to us as we construct a schedule for the split-node graph. The value $tail(v)$ is used to determine the earliest starting time for nodes with incoming edges from v, while $head(v)$ decides the latest finishing time of nodes with outgoing edges to v. Therefore, the values of $head(v)$ and $tail(v)$ alone lead to a static schedule for a SDG.

IV. SDG Scheduling Algorithms

We now discuss a formal method for scheduling a split-node graph, based on the as-early-as-possible (AEAP) scheduling algorithm [9]. For now we will assume unlimited and unrestricted resources (i.e., processors) and consider the problems of resource-constrained scheduling later.

We begin by constructing a related *sequencing graph* based on our SDG. This sequencing graph is designed to model all intra-iteration dependencies. To this end, we remove all edges from the SDG with non-zero delay count. We also replace any split node by its head and tail and re-route all zero-delay edges involving the split node. Edges leaving a split node must leave the tail of the node in the sequencing graph, and those entering a split node must now go to the head. Finally, as in the scheduling graph, a dummy source node and edges from it to all other nodes are added. The sequencing graph for Figure 1(a) is given as Figure 3(a).

We can now produce a *forward schedule* using the sequencing graph. We assume that the dummy node v_0 executes at time step zero and takes no time to execute. Since the sequencing graph is acyclic, we may apply a modified version of the $O(|V|+|E|)$ algorithm from [20] for finding the lengths of the shortest paths from v_0 to every other vertex. We begin by sorting all of the vertices, with u preceding v in the sorted list if there is an edge from u to v in the sequencing graph. Taking the vertices in sorted order, we now find the longest paths to each vertex from v_0. The length of the longest path is the starting time for the node in the first iteration; repeating this iteration gives us the complete schedule. However,

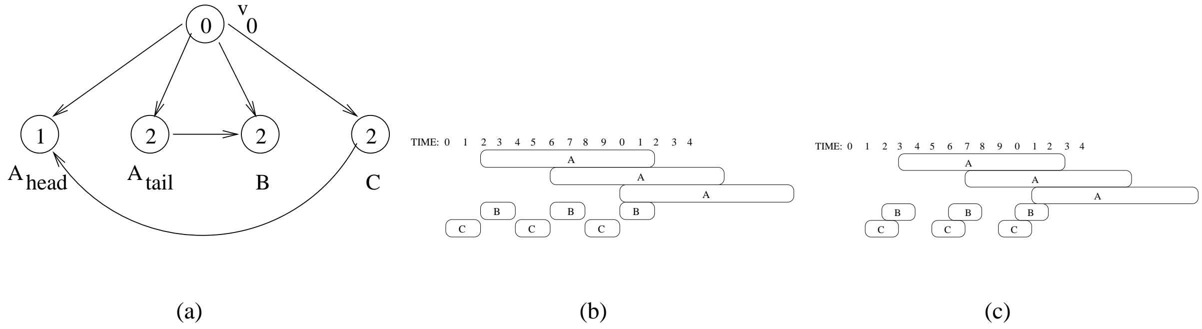


Fig. 3. (a) The sequencing graph for Figure 1(a); (b) The forward schedule for this graph; (c) The backward schedule.

since we must execute a split node in order from start to finish, we schedule only the head of any split node. This complete procedure appears as Algorithm 1.

As an example, return to the sequencing graph in Figure 3(a). The longest paths to both A_{tail} and C are zero, while those to A_{head} and B are 2. Adopting a near-optimal clock period of 4, we schedule copies of C at steps $4k$ and copies of A and B at $4k + 2$ for $k \geq 0$, as shown in Figure 3(b).

By a similar process, we may construct a SDG's *backward schedule*. First of all, when constructing the sequencing graph, we reverse the direction of all edges inherited from the original graph. (In the case of Figure 3(a), this means that the edges $A_{tail} \rightarrow B$ and $C \rightarrow A_{head}$ become the edges $B \rightarrow A_{tail}$ and $A_{head} \rightarrow C$, respectively.) With such a construction, the longest path lengths may be subtracted from the designated clock period to derive the finishing times for the nodes. We must then subtract a node's execution time to find the starting time. For example, returning to our modified version of Figure 3(a), we would compute path lengths of 0 for A_{head} and B, 1 for C and 2 for A_{tail}. Assuming again a clock period of 4, we would schedule A to start execution at step $4 - 0 - 1 = 3$, B to begin at $4 - 0 - 2 = 2$, and C to commence at $4 - 1 - 2 = 1$. This ALAP schedule is pictured in Figure 3(c).

V. Minimizing Processors in a Repeating Schedule

Having established 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:

Algorithm 1 Forward scheduling

Input: A split-node DFG $G = \langle V, E, d, t\rangle$ with clock period c

Output: An forward repeating schedule S

```
G' ← SequencingGraph(G)                          /* Construct sequencing graph for G */
for all v ∈ V' do
  time(v) ← −∞
end for
time(v_0) ← 0
/* Find longest paths to all nodes in seq. graph. */
Topologically sort the vertices of V'
for all u ∈ V' taken in sorted order do
  for all v ∈ V' adjacent to u do
    if time(v) < time(u) + t'(u) then
      time(v) ← time(u) + t'(u)
    end if
  end for
end for
for all v ∈ V do
  if v is a split node then
    S(v, 0) ← time(v_h)                          /* Schedule only the heads of split nodes. */
  else
    S(v, 0) ← time(v)                            /* Schedule the complete node if not split. */
  end if
end for
for all v ∈ V and integers i ≥ 1 do
  S(v, i) ← S(v, 0) + c · i                      /* Repeat to derive complete schedule. */
end for
```

Theorem 5.1: 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: Proven as Theorem 3.1 of [21]. ■

For instance, consider the example of Figure 1(a) again with clock period 7 and unfolding factor 2. By this result, we require $\lceil \frac{20}{7} \rceil = 3$ dedicated processors for A and $\lceil \frac{4}{7} \rceil = 1$ dedicated processor for each of B and C. In fact, each of the schedules for Figure 1(a) we have given to date reserved 5 processors for execution, one for each "row" of the schedule. However, this may be an overestimate. For example, in the forward schedule of Figure 3(b), there is no compelling reason why the iterations of B and C

cannot share a processor, thus 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 [14] and is formalized as pseudocode in [21]. It is repeated as Algorithm 2 below. 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 them so that the pieces are small enough to fit into one segment. We use an unfolding method similar to that of [15], [16], 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 ($\rho(v) =$NIL in our notation).
2) Otherwise the node overlaps segments and must be divided into parts. Let m represent the number of pieces into which the node is split. 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 $\iota(u)$ 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 $org(u)$ to identify the source in the case of HEAD or whole nodes. This entire procedure appears as Algorithm 2 below.

As an example, consider the schedule from Figure 3(b). The zeroth iterations of B and C begin and complete execution within the first 4 time steps, so are passed on to the next stage of our method as-is. On the other hand, the execution of A_0 spans the first three segments, so must be divided into 3 pieces. The HEAD section executes from time step 2 until time step 4, the BODY section from 4 to 8, and the TAIL from 8 to 12. The results of applying our algorithm to these 3 nodes, as well as a summary of their properties from the original schedule, are given in Table II.

Algorithm 2 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

Output: A set of folded vertices W

$W \leftarrow \emptyset$ /* W is the set of folded nodes */

for all $v \in V$ **do**

$m \leftarrow \left\lceil \frac{T(v)+S(v,0)}{c} \right\rceil - \left\lfloor \frac{S(v,0)}{c} \right\rfloor$

$W \leftarrow W \bigcup \{v_0, v_1, ..., v_{m-1}\}$

if $m = 1$ **then**

$\tau(v_0) \leftarrow T(v)$ /* Computation time */

$\sigma(v_0) \leftarrow S(v, 0)$ /* Start time */

$\rho(v_0) \leftarrow$NIL /* Preceding piece of folded node */

$\iota(v_0) \leftarrow$NIL /* Identification; head, body, tail or none? */

$org(v_0) \leftarrow v$ /* Corresponding original node from V */

else

$\tau(v_0) \leftarrow c - S(v, 0)$

$\sigma(v_0) \leftarrow S(v, 0)$

$\rho(v_0) \leftarrow$NIL

$\iota(v_0) \leftarrow$HEAD

$org(v_0) \leftarrow v$

for $i \leftarrow 1$ **to** $m - 2$ **do**

$\tau(v_i) \leftarrow c$

$\sigma(v_i) \leftarrow 0$

$\rho(v_i) \leftarrow v_{i-1}$

$\iota(v_i) \leftarrow$BODY

end for

$\tau(v_{m-1}) \leftarrow T(v) - \tau(v_0) - c \cdot (m - 2)$

$\sigma(v_{m-1}) \leftarrow 0$

$\rho(v_{m-1}) \leftarrow v_{m-2}$

$\iota(v_{m-1}) \leftarrow$TAIL

end if

end for

TABLE II

RESULTS OF VERTEX FOLDING FOR THE SCHEDULE OF FIGURE 3(B).

v	$S(v,0)$	$T(v)$	m		σ	τ	ρ	ι	org
A	2	10	3	A_0	2	2	NIL	HEAD	A
				A_1	0	4	A_0	BODY	NIL
				A_2	0	4	A_1	TAIL	NIL
B	2	2	1	B_0	2	2	NIL	NIL	B
C	0	2	1	C_0	0	2	NIL	NIL	C

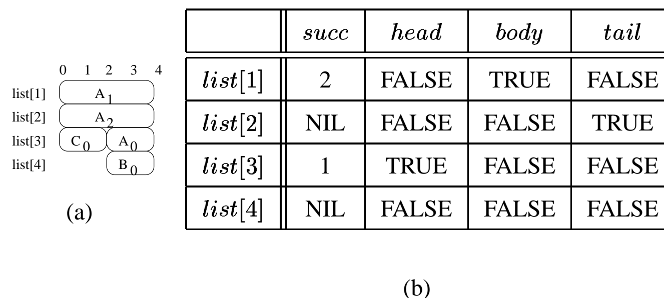
We next identify pieces that can be scheduled back-to-back on the same processor by creating *independent lists* of folded node pieces. We first sort our nodes, using starting time as the first key and "subscript" as the second. (In other words, if starting times are all equal, the HEAD of a node has 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 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 method this step is accomplished by assigning a ς value to any node which is discovered to be the ρ 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 ς 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 3 below.

Let us pick our example up where we left off, with the data in Table II. The priority queue we build from this information is $\{A_1, A_2, C_0, A_0, B_0\}$. The first three items are assigned to $list[1]$, $list[2]$ and $list[3]$, respectively, with a note made that $list[3]$ is available past time step 2. In the process, we discover two of the nodes (A_1 and A_2) with non-NIL ρ values, so must set the appropriate ς values. In this case, $\varsigma(A_0) = 1$ since $\rho(A_1) = A_0$ and A_1 was added to $list[1]$, while $\varsigma(A_1) = 2$ for similar reasons. When we pop A_0 from the queue and discover it has starting time 2, we add it to $list[3]$ rather than begin a

```
Algorithm 3 Create independent lists from folded nodes
Input: A set of folded nodes W and clock period c
Output: Independent lists of folded vertices
  /* Priority queue of vertices, ordered by σ values */
  /* first, subscript second. */
  Q ← ∅
  for all v ∈ W do
    ς(v) ←NIL
    enqueue(v, Q)
  end for
  /* Stack indep. proc. lists unassigned from time i forward */
  for i ← 1 to c do
    avail[i]← ∅
  end for
  ℓ ← 0                                              /* number of indep. lists */
  for i ← 1 to |W| do
    u ←dequeue(Q)
    if σ(u) = 0 then
      ℓ ← ℓ + 1
      list[ℓ]← u
      if ρ(u) ≠NIL and ς(ρ(u)) =NIL then
        ς(ρ(u)) ← ℓ
      end if
      push list[ℓ] onto avail[τ(u)]
    else
      if avail[σ(u)]≠ ∅ then
        /* If indep. list can hold u add it to list */
        pop list[j] from avail[σ(u)]
        list[j]←list[j]⋃{u}
        push list[j] onto avail[σ(u) + τ(u)]
      else
        ℓ ← ℓ + 1                                    /* otherwise need a new list */
        list[ℓ]← u
        push list[ℓ] onto avail[σ(u) + τ(u)]
      end if
    end if
  end for
  for i ← 1 to ℓ do
    for all u ∈list[i] do
      if σ(u) + τ(u) = c then
        succ(i) ← ς(u)
      end if
      head(i) ← head(i) or (ι(u) =HEAD)
      body(i) ← body(i) or (ι(u) =BODY)
      tail(i) ← tail(i) or (ι(u) =TAIL)
    end for

  end for
```

new list because of our note regarding that list. However, when we next pop B_0 and discover the same starting time, we no longer have a list that can receive the node and must create $list[4]$ to accommodate it. These list assignments are pictured in Figure 4(a). We next review the four lists and assign non-NIL $succ$ values to $list[1]$ and $list[3]$ since these two lists contain the two nodes (A_1 and A_0, respectively) with non-NIL ς values. Finally, we note whether any list contains HEAD, BODY or TAIL nodes. These values that we have derived are summarized in Figure 4(b).



	$succ$	$head$	$body$	$tail$
$list[1]$	2	FALSE	TRUE	FALSE
$list[2]$	NIL	FALSE	FALSE	TRUE
$list[3]$	1	TRUE	FALSE	FALSE
$list[4]$	NIL	FALSE	FALSE	FALSE

(b)

Fig. 4. Independent lists for the data from Table II: (a) the lists themselves; (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 don't. 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 unrolling 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 unrolling factor helps us keep track of which copy of a node we're actually placing. Finally, after completing a group of lists, we increment the processor count by the unrolling factor to account for pipelining, so that the next group may be assigned to the correct new processor. This procedure appears as Algorithm 4 below.

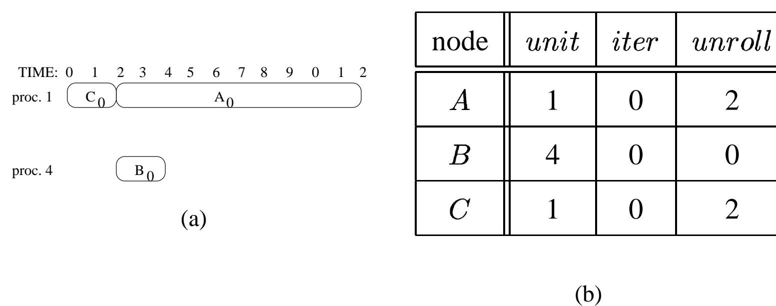
For example, apply this method to the data from Figures 4(a) and 4(b). The queue of lists without BODY or TAIL sections is $\{3, 4\}$, while those with such sections is $\{1, 2\}$. We thus begin by picking $list[3]$ and assigning it to processor 1. Since this list contains the HEAD of A, we search for the list containing the BODY of A, namely $list[1]$, the successor of the current list. We thus pull 1 from our second queue,

Algorithm 4 Use independent lists to assign processors

```
Input: Set of ℓ 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 ℓ 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 if
  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
      /* Select list without body or tail if you can */
      l ←first element of Q
    else
      l ←first element of R
    end if
    current ← ∅                          /* Current schedule piece */
    f ← 0                                /* its unrolling factor */
    while proc(l) =FALSE do
      proc(l) ←TRUE
      current ← current ⋃list[l]
      for all u ∈list[l] do
        if ι(u) =NIL or ι(u) =HEAD then
          unit(u) ← p                    /* Assign proc. p to node u iter. f */
          iter(u) ← f
        end if
      end for
      /* Select list containing body or tail following current list */
      if head(l) =TRUE or body(l) =TRUE then
        t ← l
        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
        /* For each new appended list increment unrolling factor */
        f ← f + 1
      end if
    end while
    for all u ∈ current do
      if ι(u) =NIL or ι(u) =HEAD then
        /* Assign unf. factor to all scheduled nodes */
        unroll(u) ← f
      end if
    end for
    p ← p + f

  end while
```

assign it to the current processor for execution during the next time segment, and increment the unrolling factor. Our current list, $list[1]$, contains the BODY of A, so we go back and find the location of the TAIL of A, namely $list[2]$. We pop 2, add $list[2]$ to execute on processor 1 during the third segment, and increment the unrolling factor. This new current list has no HEAD or BODY sections, so we designate nodes C and A to execute their zeroth iterations on processor 1 with unrolling factors of 2 and increment the processor count to 4. (The net effect is to reserve processors 2 and 3 for executing the first and second iterations, respectively, of these two nodes.) We now find $list[4]$ unprocessed without BODY or TAIL sections, so assign its zeroth iteration to execute on processor 4. Since it also contains no HEAD section, we increment the processor count and loop back around. This time there are no remaining unprocessed lists, so we fall out of this algorithm altogether, leaving us with the initial processor assignment in Figure 5(a) and the values for each node listed in the table in Figure 5(b).



node	$unit$	$iter$	$unroll$
A	1	0	2
B	4	0	0
C	1	0	2

(b)

Fig. 5. The initial processor assignment derived from the data in Figures 4(a) and 4(b).

Finally, we must propagate this initial processor assignment throughout the remaining iterations of our nodes. Previously we assigned a processor for a designated iteration of each node. We also recorded an unrolling factor f, telling us that $f + 1$ copies of this particular node are in the pipeline simultaneously. It is now a simple task to assign the $f + 1$ copies of the node to the $f + 1$ processors we reserved for these tasks, then repeat this assignment statically to infinity, thus completing our processor assignment. This final step in the method is summarized in Algorithm 5.

We now complete our example, using the values from Figure 5(b). According to this information, iteration 0 of A is assigned to processor ($unit$) 1. Its $unroll$ value of 2 then tells us that the next two iterations of A are executing simultaneously on the next two processors; hence A_1 is assigned to processor 2 and A_2 to processor 3. We now repeat to infinity, with A_{3k+i} being assigned to processor $i + 1$ for all $k \geq 0$ and $i = 0, 1, 2$. The values for C are the same, so the processor assignment is the same. Finally,

Algorithm 5 Expand iteration zero processor assignment to complete processor assignment

Input: Set of folded nodes W with assigned $unit$ and $unroll$ values

Output: A complete processor assignment

for all $u \in W$ **do**
  if $\iota(u)$ =NIL **or** $\iota(u)$ =HEAD **then**
    /* Assign first few iterations */
    for $i \leftarrow 0$ to $unroll(u)$ **do**
      $P(org(u), (iter(u) + i) \mod (unroll(u) + 1)) \leftarrow unit(u) + i$
    end for
    /* Repeat first few iterations to infinity */
    for all $i \geq 0$ **do**
      $P(org(u), i + unroll(u) + 1) \leftarrow P(org(u), i)$
    end for
  end if
end for

node B has its zeroth iteration assigned to processor 4 and has no $unroll$ value. Thus the execution of B's iterations is not pipelined, with all assigned to execute on processor 4. We thus derive the final time- and processor-optimal schedule seen in Figure 6.

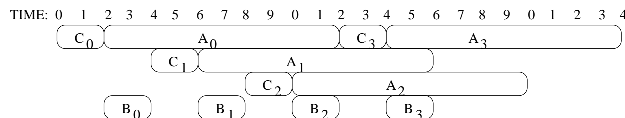


Fig. 6. Final processor-optimal schedule for first example.

VI. UNFOLDING SPLIT-NODE GRAPHS

While our scheduling algorithm appears effective, this first example demonstrates its weakness. Because our algorithms require an integral clock period, the best we could accomplish with our formal methods was to create a schedule for Figure 1(a) with a near-optimal period of 4. As noted, the iteration bound for Figure 1(a) is $\frac{7}{2}$. If we unfold this graph twice and then apply our scheduling algorithms with a

clock-period of 7, we would achieve rate-optimality. However, this assumes that we are able to unfold a split-node graph, a problem that has yet to be explored.

Unfolding has been discussed extensively in [17], [19]. Briefly, if f is a positive integer, we wish to alter our graph so that f consecutive *iterations* (i.e., executions of all of a DFG's tasks) are visible simultaneously. To do this, we create f copies of each node, replacing node u in the original graph by the nodes u_1 through u_f in our new graph. This process is known as *unfolding* the graph G f times and results in the *unfolded graph* $G_f = \langle V_f, E_f, d_f, t_f \rangle$. The vertex set V_f is simply the union of the f copies of each node in V. Since they are all exact copies, the computation times remain the same, i.e. $t_f(u_f) = t(u)$ for every copy u_f of $u \in V$. Each edge of G also corresponds to f copies in the unfolded graph. However, the delay counts of the copies do not match that of the original edge. As demonstrated in [1], an edge (u_i, v_j) having d delays in the unfolded graph represents a precedence relation between node u in the ith iteration and node v in iteration $d \cdot f + j$ in the original graph.

There is one other new construction we will require for our upcoming discussion. We see in Figure 1(a) that we wish to split node A into four pieces while the other nodes remain whole. If we split A into four separate nodes with appropriate computation times, as shown in Figure 7(b), the resulting graph is functionally similar to the original but can be manipulated via traditional methods, allowing us to gain insight while we study our split-node graph. We will call such a graph an *extended graph*.

The properties for an unfolded graph without split nodes are outlined in [19]. Each of these results may be extended to paths in such a graph, specifically to an extended graph which corresponds to a split-node graph. Thus, our basic strategy will be to apply our original theory to an extended graph in order to derive the desired properties for the split-node graph. Proceeding in this fashion produces these results:

Theorem 6.1: Let $G = \langle V, E, d, t \rangle$ be a split-node graph. Let $u, v \in V$ and $e = (u, v) \in E$. Let f be an unfolding factor.

1) Let u be a split node. For $i = 0, 1, ..., f-1$, the N delays in u_i lie between that node's copies of pieces $f-i-1$ and $f-i$, $2f-i-1$ and $2f-i$,..., $Nf-i-1$ and $Nf-i$.
2) The ith delay of node u lies in node u_j of the unf olded graph where $j \equiv (Nf - i - 1) \mod f$.
3) If u is a split node containing N delays, then u_i contains $\left\lfloor \frac{N+i}{f} \right\rfloor$ delays for $i = 0, 1, ..., f-1$.
4) For any integers $0 \le i, j < f$, there is an edge $e_f = (u_i, v_j)$ in the unfolded graph G_f if and only if $d^+(u \to v) = d_f^+(u_i \to v_j) \cdot f + j - i$.

5) For $0 \leq i, j < f$ with $j - i \equiv d^+(u \to v) \bmod f$, there is an edge $e_f = (u_i, v_j)$ with delay count

$$d_f^+(u_i \to v_j) = \begin{cases} \left\lfloor \frac{N+d(e)}{f} \right\rfloor & if\ j \geq i \\ \\ \left\lceil \frac{N+d(e)}{f} \right\rceil & otherwise \end{cases}$$

where N is the number of delays contained within the split node u.

6) The f copies of edge $e = (u, v)$ in G_f are the edges $e_i = (u_i, v_{(i+N+d(e)) \bmod f})$ for $i = 0, 1, ..., f-1$.

7) The total number of delays along the f copies of edge e, not including delays contained within the end nodes, is $d(e)$; in other words, $d(e) = \sum_{i=0}^{f-1} d_f(e_i)$.

Proof: Proven as Theorem 3.1 of [22]. ■

Having established the needed properties, we outline the unfolding method formalized as Algorithm 6 below. Broadly speaking, we first distribute delays within source nodes. The simplest way to do this is to create f copies of each node, each of which contains all delays, then remove the correct delays from within each node. Next, we assign delays to each edge between nodes. Note that we have already accounted for delays within end nodes and need only concern ourselves with delays along the actual edges. Here we modify the algorithm from [19], taking care to adjust our figures by subtracting delays within source nodes.

VII. EXAMPLE

To review our methods, let us reconsider our previous sample graph (repeated as Figure 7(a) below) unfolded by a factor of 2. Our unfolding procedure takes place in two stages.

First we make two copies of the node set and assign delays within nodes. Node A is the only split node in the graph, so we are only concerned with dividing the three delays among A_0 and A_1. As we have previously specified, delay number zero (between the pieces of A with computation times 1 and 4) is removed from A_0, delay one (between the pieces of A with times 4 and 3) is taken away from A_1, and the remaining delay disappears from A_0. The result, displayed in Figure 7(c), is node A_0 split in half and node A_1 divided into pieces with computation times 1, 7 and 2. Let $\Delta(v)$ be the number of delays contained within node v so that $\Delta(A_0) = 1$ and $\Delta(A_1) = 2$ in this case.

Next we assign copies of the four edges with appropriate delays counts.

1) First we consider the edge from A to B. There are no delays along the actual edge, but there are three delays within the source node. Thus the variables δ and ρ in our algorithm are both one. We

Algorithm 6 Unfolding a split-node graph

Input: A SDG $G = \langle V, E, d, t \rangle$, an integer f

Output: The unfolded graph $G_f = \langle V_f, E_f, d_f, t_f \rangle$

```
for all nodes u ∈ V containing Δ(u) delays do
  for i = 0 to f − 1 do
    Add a copy of node u as u_i to V_f
    Δ(u_i) ← Δ(u)
  end for
  for i = 0 to Δ(u) − 1 do
    for j = 0 to f − 1 do
      if j ≢ (−i − 1) mod f then
        Remove the i^th delay from node u_{i mod f}
        Δ(u_{i mod f}) ← Δ(u_{i mod f}) − 1
      end if
    end for
  end for
end for
for all edges e = (u, v) in E do
  δ ← (Δ(u) + d(e)) mod f
  ρ ← ⌊(Δ(u)+d(e)) / f⌋
  for i = 0 to f − δ − 1 do
    Add edge e_f = (u_i, v_{i+δ}) to E_f
    d_f(e_f) ← ρ − Δ(u_i)
  end for
  for i = f − δ to f − 1 do
    Add edge e_f = (u_i, v_{i+δ−f}) to E_f
    d_f(e_f) ← ρ + 1 − Δ(u_i)
  end for
end for
```

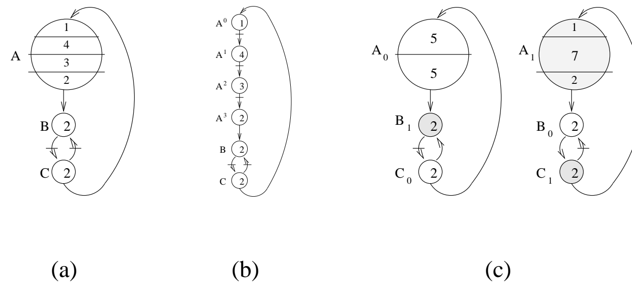


Fig. 7. (a) Our original example; (b) Its extended graph; (c) The graph unfolded by a factor of 2.

therefore add the edge (A_0, B_1) with delay count $\rho - \Delta(A_0) = 1 - 1 = 0$ during the first loop, and (A_1, B_0) with $\rho - \Delta(A_1) + 1 = 1 - 2 + 1 = 0$ when in the second.

2) Next we deal with the edge (B, C) containing one delay on the edge but none within the source node. Hence $\delta = 1$ and $\rho = 0$ in this case, and since $\Delta(B_i) = \Delta(C_i) = 0$ for $i = 0, 1$, we add a zero-delay edge (B_0, C_1) in the first loop and a one-delay edge (B_1, C_0) in the second.
3) Similarly, the one-delay edge (C, B) spawns the zero-delay edge (C_0, B_1) in the first loop and the one-delay edge (C_1, B_0) in the second.
4) Finally, the zero-delay edge (C, A) must be handled. Since $\delta = \rho = 0$, we never execute the second loop. Two passes of the first loop are executed which produce two zero-delay edges, (C_0, A_0) then (C_1, A_1).

The final result appears in Figure 7(c). For clarity, the nodes comprising iteration one are shaded.

With unfolding complete, we now attempt to schedule the unfolded graph with a clock period of 7 time units. Applying our methods yields the final rate- and processor-optimal schedule for Figure 1(a) pictured in Figure 8. Note that this schedule differs from that of Figure 1(b), which is also rate-optimal.

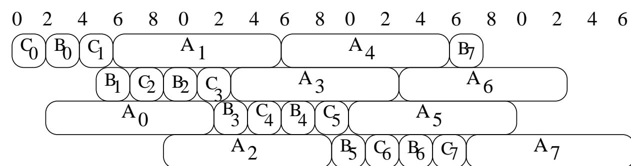


Fig. 8. Rate- and processor-optimal schedule for Figure 1(a).

Finally, let us consider the summary of our experiments in Table III and observe the effectiveness

of our methods. Prior to the work of the last two sections, the best estimate for hardware required by a schedule resulting from DFG scheduling is the figure arrived at by Theorem 5.1. These figures for our experiments appear in Table III under the "Before" column. The processor count resulting from our minimization algorithms is given in this table under "After", with the percent reduction in hardware appearing in the last column. In addition to noting the effectiveness of our algorithm, these figures point out the problem of unfolding. While it reduces the overall iteration period, it greatly increases the size of our graph and, consequently, the processor requirements of a resulting schedule as estimated by Theorem 5.1.

TABLE III

COMPARISON OF SCHEDULES BEFORE AND AFTER PROCESSOR OPTIMIZATION

	Clock Period	No. Processors		Percent Reduced
		Before	After	
Fig. 1(a) without unfolding	4	5	4	20
Fig. 1(a) unfolded twice	7	8	4	50
Figure 3.12(a) of [23]	3	8	5	37.5
Fig. 4(a) of [22] unfolded thrice	11	9	3	66.7

VIII. 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 explored the properties of the particular idealized form of split-node graphs constructed by the methods of [7], [8]. We have developed scheduling algorithms for split-node graphs, and then discussed a way to reduce the hardware requirements of such schedules. 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. We have also constructed an unfolding algorithm for split-node graphs and combined it with our scheduling methods to achieve rate-optimality in all cases. Finally, we have demonstrated our methods on specific examples.

While these methods are good, there is currently nothing to compare them with. The proposed scheduling method proceeds in two parts: first a time schedule is constructed, then the minimal processor assignment derived to fit the schedule. Methods exist for scheduling regular DFGs which perform both

of these steps simultaneously at a much lower cost. We hope to modify such methods to fit this new model very soon.

REFERENCES

[1] L.-F. Chao and E. H.-M. Sha, "Scheduling data-flow graphs via retiming and unfolding," *IEEE Trans. Parallel Distrib. Syst.*, vol. 8, pp. 1259–1267, 1997.

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

[3] S. Kung, J. Whitehouse, and T. Kailath, *VLSI and Modern Signal Processing*. Prentice Hall, 1985.

[4] L.-F. Chao and E. H.-M. Sha, "Retiming and unfolding data-flow graphs," in *Proc. Int. Conf. Parallel Processing*, 1992, pp. II 33–40.

[5] T. O'Neil, S. Tongsima, and E. H.-M. Sha, "Extended retiming: Optimal retiming via a graph-theoretical approach," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, vol. 4, 1999, pp. 2001–2004.

[6] T. O'Neil and E. H.-M. Sha, "Rate-optimal graph transformation via extended retiming and unfolding," in *Proc. IASTED 11th Int. Conf. Parallel & Distrib. Comput. & Syst.*, vol. 10, 1999, pp. 764–769.

[7] T. O'Neil, S. Tongsima, and E. H.-M. Sha, "Optimal scheduling of data-flow graphs using extended retiming," in *Proc. ISCA 12th Int. Conf. Parallel & Distrib. Comput. Syst.*, 1999, pp. 292–297.

[8] T. O'Neil and E. H.-M. Sha, "Optimal graph transformation using extended retiming with minimal unfolding," in *Proc. IASTED 12th Int. Conf. Parallel & Distrib. Comput. & Syst.*, vol. I, 2000, pp. 128–133.

[9] G. DeMicheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, Inc., 1994.

[10] L.-F. Chao and E. H.-M. Sha, "Static scheduling for synthesis of DSP algorithms on various models," *J. VLSI Signal Processing*, vol. 10, pp. 207–223, 1995.

[11] F. Gasperoni and U. Schwiegelshohn, "Generating close to optimum loop schedules on parallel processors," *Parallel Processing Lett.*, vol. 4, pp. 391–403, 1994.

[12] ——, "Transforming cyclic scheduling problems into acyclic ones," in *Scheduling Theory and Its Applications*. John Wiley & Sons, 1995, pp. 241–258.

[13] C. Hanen and A. Munier, "Cyclic scheduling on parallel processors: An overview," in *Scheduling Theory and Its Applications*. John Wiley & Sons, 1995, pp. 194–226.

[14] Y.-C. Ho and J.-C. Tsay, "Fully static processor-optimal assignment of data-flow graphs," *IEEE Signal Processing Lett.*, vol. 4, pp. 146–148, 1997.

[15] K. Ito, L. Lucke, and K. Parhi, "Module selection and data format conversion for cost-optimal DSP synthesis," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, 1994, pp. 322–329.

[16] L. Lucke and K. Parhi, "Generalized ILP scheduling and allocation for high-level DSP synthesis," in *Proc. IEEE Custom Integrated Circuits Conf.*, 1993, pp. 5.4.1–5.4.4.

[17] K. Parhi and D. Messerschmitt, "Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding," *IEEE Trans. Comput.*, vol. 40, pp. 178–195, 1991.

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

[19] L.-F. Chao, "Scheduling and behavioral transformations for parallel systems," Ph.D. dissertation, Dept. Comput. Sci., Princeton Univ., 1993.

[20] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*. McGraw-Hill, Inc., 1991.

[21] T. O'Neil and E. H.-M. Sha, "Minimizing resources in a repeating schedule for a split-node data-flow graph," in *Proc. IEEE/ACM Great Lakes Symp. VLSI*, 2002, pp. 136–141.

[22] ——, "Unfolding a split-node data-flow graph," in *Proc. IASTED 14th Int. Conf. Parallel & Distrib. Comput. & Syst.*, 2002, pp. 717–722.

[23] T. O'Neil, "Techniques for optimizing loop scheduling," Ph.D. dissertation, Dept. Comput. Sci. & Eng., Univ. Notre Dame, 2002.