

Unfolding a Split-Node Data-Flow Graph

Timothy W. O’Neil
Computer Science Department
University of Akron
Akron, OH 44325-4002
email: toneil@cs.uakron.edu

Edwin H.-M. Sha
Computer Science Department
Univ. of Texas at Dallas
Richardson, TX 75083-0688
email: 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, with it implied that we are able to unfold such a graph. Generalizing the known unfolding methods so that they apply to this new model has heretofore not been explored. In this paper, we logically prove the specific changes that transpire when a split-node graph is unfolded and propose an algorithm for performing unfolding.

KEY WORDS

Parallel and Distributed Systems, Compilers, Optimization, Modeling Languages

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) [3]. 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 [5], digital signal processing [4] and program descriptions [2].

The application of graph transformations like *unfolding* [11] to DFGs in order to reduce execution times has been widely noted. In our previous work [6,7,9,10], 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*, with it implied that we are able to unfold such a graph.¹ An unfolding algorithm for

this generalized data-flow model has not been heretofore proposed. In this paper, we will rectify this situation by constructing the necessary methodology.

In this paper, we formally define a split-node data-flow graph and some associated terminology. We logically prove the specific changes that transpire when a split-node graph is unfolded and propose an algorithm for performing unfolding. Finally, we demonstrate our methods on specific examples.

2 Background

In this section, we wish to present the definitions and results relating to unfolding and unfolded graphs. We will rely on this previously-presented background material [1,8,11] heavily as we establish our new results.

2.1 Unfolding and Unfolded Graphs

Recall that a *data-flow graph* (DFG) is a finite, directed, weighted graph $G = \langle V, E, d, t \rangle$ where V is a set of computation nodes, E is a set of edges between nodes, $d : E \rightarrow \mathbf{N}$ is a function representing the delay count of each edge, and $t : V \rightarrow \mathbf{N}$ representing the computation time of each node. Defining an *iteration* to be one execution of all tasks in a DFG, then delays along an edge represent precedence relations across iterations.

Now, let f be a positive integer. We wish to alter our graph so that f consecutive iterations 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.

¹This 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.

As an example, suppose that we want to unfold the graph in Figure 1(a) by a factor of 3. The five edges without delays represent precedence relations within each iteration of the graph, and are passed as they are to the unfolded graph displayed in Figure 1(b). On the other hand, edges with delays represent dependencies between iterations. Let u_i be the occurrence of node u in the i^{th} iteration and investigate the two edges in our graph which have delays:

- The edge (D, A) having delay count three tells us that D_0 must precede A_3 , D_1 precedes A_4 , D_2 precedes A_5 , and so on. Since we are unfolding our graph by a factor of 3, the nodes D_0, D_1, D_2, A_0, A_1 and A_2 are all present in our graph. Thus, when we execute this graph instead of the original one, what was the execution of A in iteration 3 is now the execution of A_0 in iteration 1. Therefore, we now need a one-delay edge from D_0 to A_0 . Similarly, we require one-delay edges (D_1, A_1) and (D_2, A_2) as well. As we can see here, unfolding is permitting us to replace three delays on the original edge with one on each of the unfolded edges. In general, f delays on an edge in graph G are represented by 1 delay on each copy of the edge in the unfolded graph G_f .
- Similarly, the two-delay edge (E, C) tells us that E_0 precedes C_2 , E_1 precedes C_3 , and E_2 precedes C_4 . It is simple to insert the edge (E_0, C_2) since both nodes are present in the unfolded graph. As before, C_0 in iteration 1 is actually the original C_4 , so this relation can be indicated by the edge (E_1, C_0) with 1 delay. Similarly, we also have a 1-delay edge (E_2, C_1) , as can be seen in Figure 1(b).

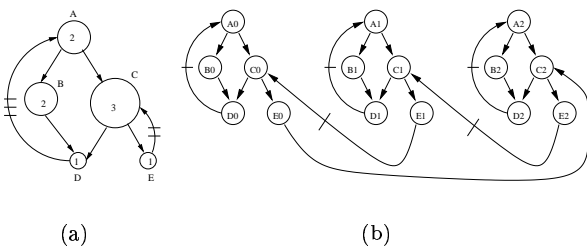


Figure 1. (a) A sample DFG; (b) This DFG unfolded by a factor of 3

As we can see, an edge (u_i, v_j) having d delays in the unfolded graph represents a precedence relation between node u in the i^{th} iteration and node v in iteration $d \cdot f + j$ in the original graph. This idea is formalized in the following theorem, which is proven in [3].

Theorem 2.1 Let $e = (u, v)$ be an edge in DFG G . Let f be an unfolding factor for G . Then:

1. For all $i, j \in \{0, 1, 2, \dots, f - 1\}$, there exists an edge $e_f = (u_i, v_j)$ in G_f if and only if $d(e) = d_f(e_f) \cdot f + j - i$.
2. For all integers $i, j \in \{0, 1, 2, \dots, f - 1\}$ with $j \equiv (i + d(e)) \pmod f$, there exists an edge $e_f = (u_i, v_j)$ in G_f with $d_f(e_f) = \lfloor \frac{d(e)}{f} \rfloor$ if $i \leq j$ and $\lceil \frac{d(e)}{f} \rceil$ otherwise.
3. The f copies of edge e in G_f are the edges $e_i = (u_i, v_{(i+d(e)) \pmod f})$ for $i = 0, 1, 2, \dots, f - 1$.
4. The total number of delays of the f copies of edge e is $d(e)$, i.e. $d(e) = \sum_{i=0}^{f-1} d_f(e_i)$.

2.2 Extended Retiming

As in [6, 7], an *extended* (or *f-extended*) *retiming* of a DFG $G = \langle V, E, d, t \rangle$ is a function $r : V \rightarrow \mathbf{Z} \times \mathbf{Q}^f$ where, for all $v \in V$, $r(v) = i + \left(\frac{r_1}{t(v)}, \frac{r_2}{t(v)}, \dots, \frac{r_f}{t(v)} \right)$ for some integers i, r_1, r_2, \dots, r_f where $0 \leq r_k < t(v)$ for $k = 1, 2, \dots, f$. We view the integer constant i as the number of delays that are pushed to each outgoing edge of v , while the f -tuple lists the positions of delays within the node v . Note that a value of zero within the f -tuple is merely a placeholder used to simplify our notation; we cannot have a delay at this position. For example, consider the graph of Figure 2(a). A 3-extended retiming with $r(A) = 1 + \frac{1}{10}(1, 5, 8)$, $r(B) = 1$ and $r(C) = 0$ results in the retimed graph of Figure 2(b).

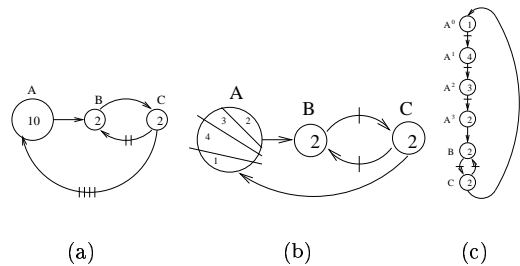


Figure 2. (a) A sample DFG; (b) This DFG retimed; (c) Its extended graph.

In [6] we demonstrated the effectiveness of our extended retiming transformation via several experiments, getting an optimal iteration period while requiring less unfolding in all cases. The usefulness of this new transformation is clear, but interpreting the split-node graph that results from its application still

presents a problem. Furthermore, there exist examples where combining unfolding with extended retiming yields a more efficient design. (We have noted such an example in [8].) The problem of unfolding data-flow graphs is not new but the model is. In using a split-node DFG to represent a situation, we are conveying additional information about the underlying properties of the entity being modeled. We must carefully account for this additional data and use it to make specific modifications to our existing unfolding techniques to derive a more general algorithm which applies to our situation.

2.3 Other Notations

We have just defined several concepts which we will need for this discussion. In addition to reviewing these previous ideas, we must augment our existing theory slightly.

We define a *split-node data-flow graph (SDG)* with splitting degree δ to be a finite, directed, weighted graph $G = \langle V, E, d, t \rangle$ where V is the vertex set; $E \subseteq V \times V$ is the edge set, representing precedence relations among the nodes; $d : E \rightarrow \mathbf{Z}$ is a function with $d(e)$ the delay count for edge e ; and $t : V \rightarrow \mathbf{Z}^\delta$ 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. (Trivially if $\delta = 1$ the SDG is simply a data-flow graph as defined previously.) If a node v is not split $t(v)$ is an integer rather than a δ -tuple. For example, in the SDG of Figure 2(b), $t(A) = (1, 4, 3, 2)$ 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. As we have stated, 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 2(b) 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. In the case of Figure 2(b) we can build a schedule in such a way that 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 .

Given an edge $e = (u, v)$ in a data flow graph G , we use $d(u \rightarrow v)$ to denote the total number of delays along an edge, including delays contained within the end nodes u and v . However, we will refer to the number of delays on the edge *not including* delays within end nodes as $d(e)$ as in the traditional case. As an example, consider the graph in Figure 2(b). Denote the edges (A, B) , (B, C) , (C, B) and (C, A) as e_1 , e_2 , e_3 and e_4 , respectively. Then $d(A \rightarrow B)$ and $d(C \rightarrow A)$ are each 3 due to the split end node, even though $d(e_1)$ and $d(e_4)$ are each zero. On the other hand, $d(B \rightarrow C) = d(C \rightarrow B) = d(e_2) = d(e_3) = 1$ since there is no split end node for these two edges.

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 our example again, we see that $d^+(A \rightarrow B) = d^-(C \rightarrow A) = 3$ but $d^-(A \rightarrow B) = d^+(C \rightarrow A) = 0$. It is easy to see that $d(u \rightarrow v) = d^+(u \rightarrow v) + d^-(u \rightarrow v) - d(e)$ for any edge $e = (u, v)$.

We see in Figure 2(b) 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 2(c), 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*. If G is our original graph, we will designate the extended graph derived from G as X^G .

3 Unfolding Split-Node Graphs

We will now apply our existing knowledge to the study of unfolding graphs with split nodes. We begin by establishing characteristics of the unfolded graph, followed by the derivation of an algorithm for unfolding a split-node graph.

3.1 Properties of Unfolded Graphs

We have already presented many properties for an unfolded graph without split nodes. 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 3.1 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 node split by N delays into $N+1$ pieces. For $i = 0, 1, \dots, f-1$, the N_i delays within node u_i in the unfolded graph lie between that node's copies of consecutive pieces $f-i-1$ and $f-i$, $2f-i-1$ and $2f-i$, ..., $N_i f-i-1$ and $N_i f-i$.
2. The i^{th} delay of node u lies in node u_j of the unfolded graph where $j \equiv (Nf-i-1) \pmod f$, where N is the number of delays contained within u .
3. If u is a split node containing N delays, then u_i contains $\lfloor \frac{N+i}{f} \rfloor$ delays for $i = 0, 1, \dots, f-1$.
4. For any integers $0 \leq i, j < f$, there is an edge $e_f = (u_i, v_j)$ in the unfolded graph G_f if and only if $d^+(u \rightarrow v) = d_f^+(u_i \rightarrow v_j) \cdot f + j - i$.
5. For $0 \leq i, j < f$ with $j - i \equiv d^+(u \rightarrow v) \pmod f$, there is an edge $e_f = (u_i, v_j)$ with delay count

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

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 $e_i = (u_i, v_{(i+N+d(e)) \pmod f})$ for $i = 0, 1, \dots, f-1$.
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:

1. Designate the N pieces of u by u^0, u^1, \dots, u^{N-1} and consider the unfolded extended graph X_f^G . In the original extended graph, u^i was separated from u^{i+1} by an edge containing one delay for $i = 0, \dots, N-2$. Therefore, in X_f^G , there are zero-delay edges from u_j^i to u_{j+1}^{i+1} for $i = 0, \dots, N-2$ and $j = 0, \dots, f-2$, along with one-delay edges from u_{j-1}^i to u_0^{i+1} for $i = 0, \dots, N-2$. We now begin constructing G_f from X_f^G by reassembling each split node u_i , $0 \leq i \leq f-1$, from the $N \cdot f$ pieces of the copies of u . The first part of u_i consists of all subnodes along the path $u_i^0, u_{i+1}^1, \dots, u_{f-1}^k$ for some integer k . By matching the upper and lower indices on the subnodes we see that $k-0 = f-1-i$, or $k = f-i-1$. We then have the one-delay edge between u_{f-1}^{f-i-1} and u_0^{f-i} which inserts the first

delay into u_i before starting another zero-delay path between u_0^{f-i} and u_{f-1}^{2f-i-1} . Following this pattern, we see that we are inserting another delay into u_i every f subnodes until we have assigned all delays.

2. If the i^{th} delay lies in u_j then there exists an integer k such that $kf - j - 1 = i$ from (1) above. Thus $j = kf - i - 1 \equiv (Nf - i - 1) \pmod f$.
3. We must insert all delays into u_i before we run out of pieces. Thus the position of the last delay must be smaller than the total number of pieces, or $fn - i \leq N$. This yields $n \leq \frac{N+i}{f}$, and since n must be integral, we maximize it by setting it equal to the floor of this fraction.
4. First note that edge e in G corresponds to a path from u^0 to v^0 in X^G which consists of the edges $e_k = (u^k, u^{k+1})$ for $k = 0, 1, \dots, N-1$ and $e_N = (u^N, v^0)$. Similarly, there is an edge (u_i, v_j) in G_f if and only if there is a path from u_i^0 to v_j^0 in X_f^G , which happens if and only if X_f^G contains the edges $\epsilon_k = (u_{(i+k) \pmod f}^k, u_{(i+k+1) \pmod f}^{k+1})$ for $k = 0, 1, \dots, N-1$ and $\epsilon_N = (u_{(i+N) \pmod f}^N, v_j^0)$. By Theorem 2.1(1), this occurs if and only if

$$\begin{cases} d(e_k) = d_f(\epsilon_k) \cdot f + (i+k+1) \pmod f \\ \quad \quad \quad - (i+k) \pmod f \\ d(e_N) = d_f(\epsilon_N) \cdot f + j - (i+N) \pmod f \end{cases},$$

for $0 \leq k \leq N-1$, which is equivalent to

$$\begin{aligned} d^+(u \rightarrow v) &= \sum_{k=0}^N d(e_k) \\ &= f \cdot \sum_{k=0}^N d_f(\epsilon_k) + j \\ &\quad + \sum_{k=0}^{N-1} (i+k+1) \pmod f \\ &\quad - \sum_{k=0}^{N-1} (i+k) \pmod f \\ &= f \cdot d_f^+(u_i \rightarrow v_j) + j \\ &\quad + \sum_{k=1}^N (i+k) \pmod f \\ &\quad - (i + \sum_{k=1}^N (i+k) \pmod f) \\ &= f \cdot d_f^+(u_i \rightarrow v_j) + j - i \end{aligned}$$

5. As noted above, the edge e in G corresponds to the path from u^0 to v^0 in X^G . By the analogue of Theorem 2.1(2) for paths, under these conditions there is a path from u_i^0 to v_j^0 in X_f^G with delay count

$$D_f(u_i^0 \Rightarrow v_j^0) = \begin{cases} \lfloor \frac{D(u^0 \Rightarrow v^0)}{f} \rfloor & \text{if } j \geq i \\ \lfloor \frac{D(u^0 \Rightarrow v^0)}{f} \rfloor & \text{otherwise} \end{cases}.$$

Since $D_f(u_i^0 \Rightarrow v_j^0) = d_f^+(u_i \rightarrow v_j)$ and $D(u^0 \Rightarrow v^0) = d^+(u \rightarrow v) = N + d(e)$ the result follows immediately.

6. By logic similar to the previous section, the analogue of Theorem 2.1(3) for paths implies the existence of f copies of this path in X_f^G , specifically the paths from u_i^0 to $v_{(i+D(u^0 \Rightarrow v^0)) \bmod f}$ for $i = 0, 1, \dots, f-1$ and our desired property clearly follows.

7. We have seen that $d_f^+(u_i \rightarrow v_j) = \frac{1}{f}(d^+(u \rightarrow v) + i - j) = \frac{1}{f}(N + d(e) + i - j)$, and so

$$\begin{aligned} & \sum_{i=0}^{f-1} d_f^+(u_i \rightarrow v_{(i+N+d(e)) \bmod f}) \\ &= N + d(e) \\ &+ \frac{1}{f} \left(\sum_{i=0}^{f-1} i - \sum_{i=0}^{f-1} (N + i + d(e)) \bmod f \right) \\ &= N + d(e) + \frac{1}{f} \left(\sum_{i=0}^{f-1} i - \sum_{i=0}^{f-1} i \right) \\ &= N + d(e). \end{aligned}$$

Since there are N total delays distributed among the f copies of node u , $\sum_{i=0}^{f-1} d_f(e_i) = \sum_{i=0}^{f-1} d_f^+(u_i \rightarrow v_{(i+N+d(e)) \bmod f}) - N = d(e)$ and our result is shown. \square

3.2 Our Unfolding Algorithm

Having established the needed properties, we formalize our unfolding method as Algorithm 1 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 [1], taking care to adjust our figures by subtracting delays within source nodes.

4 Examples

To observe Algorithm 1 in action, we shall apply it to a couple of examples.

4.1 First Example

To begin, let us reconsider our previous sample graph (repeated as Figure 3(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

Algorithm 1 Unfolding a split-node graph

Input: A DFG $G = \langle V, E, d, t \rangle$, an integer f
Output: The unfolded graph $G_f = \langle V_f, E_f, d_f, t_f \rangle$

```

for all nodes  $u \in V$  containing  $\Delta(u)$  delays do
  for  $i = 0$  to  $f - 1$  do
    Add a copy of node  $u$  as  $u_i$  to  $V_f$ 
     $\Delta(u_i) \leftarrow \Delta(u)$ 
  end for
  for  $i = 0$  to  $\Delta(u) - 1$  do
    for  $j = 0$  to  $f - 1$  do
      if  $j \not\equiv (-i - 1) \bmod f$  then
        Remove the  $i^{\text{th}}$  delay from node  $u_i \bmod f$ 
         $\Delta(u_i \bmod f) \leftarrow \Delta(u_i \bmod f) - 1$ 
      end if
    end for
  end for
end for
for all edges  $e = (u, v)$  in  $E$  do
   $\delta \leftarrow (\Delta(u) + d(e)) \bmod f$ 
   $\rho \leftarrow \lfloor \frac{\Delta(u) + d(e)}{f} \rfloor$ 
  for  $i = 0$  to  $f - \delta - 1$  do
    Add edge  $e_f = (u_i, v_{i+\delta})$  to  $E_f$ 
     $d_f(e_f) \leftarrow \rho - \Delta(u_i)$ 
  end for
  for  $i = f - \delta$  to  $f - 1$  do
    Add edge  $e_f = (u_i, v_{i+\delta-f})$  to  $E_f$ 
     $d_f(e_f) \leftarrow \rho + 1 - \Delta(u_i)$ 
  end for
end for

```

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 3(b), is node A_0 split in half and node A_1 divided into pieces with computation times 1, 7 and 2. We note here that $\Delta(A_0) = 1$ and $\Delta(A_1) = 2$ in this case.

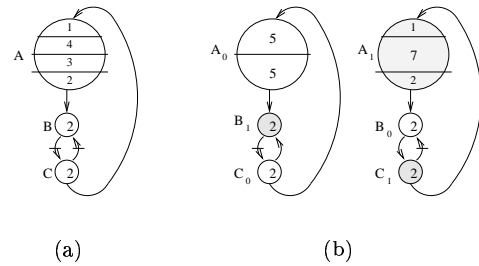


Figure 3. (a) Our original example; (b) The graph unfolded by a factor of 2.

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 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 3(b). For clarity, the nodes comprising iteration one are shaded.

4.2 Second Example

Next, consider the SDG in Figure 4(a) which we wish to unfold three times. Node A is the only split node, containing two delays. By our algorithm, the zeroth delay is taken away from all copies of A except A_2 , while the first delay disappears from all copies except A_1 . Because the edge (A, B) contains two delays within its source node, we require edges (A_0, B_2) , (A_1, B_0) and (A_2, B_1) . These last two edges require one delay each, a demand satisfied by the delays contained within the source nodes of these edges. (B, C) in the original graph contains one delay, giving us zero-delay edges (B_0, C_1) and (B_1, C_2) and the one-delay edge (B_2, C_0) . Finally the zero-delay edge (C, A) gives us three zero-delay edges in the unfolded graph, and the unfolded graph as it appears in Figure 4(b) is complete.

5 Conclusion

In this paper, we have formally defined a split-node data-flow graph and redefined the standard terminology to fit this new paradigm. We have logically proven the specific changes that transpire when a split-node graph is unfolded and propose an algorithm for performing unfolding. Finally, we have demonstrated our methods on specific examples.

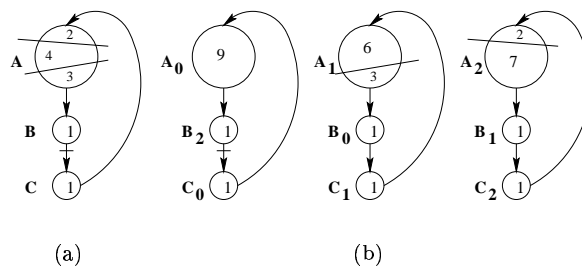


Figure 4. (a) Another sample SDG; (b) Figure 4(a) unfolded thrice.

Acknowledgement

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

References

- [1] L.-F. Chao. *Scheduling and Behavioral Transformations for Parallel Systems*. PhD thesis, Dept. of Comput. Sci., Princeton Univ., 1993.
- [2] L.-F. Chao and E. H.-M. Sha. Retiming and unfolding data-flow graphs. In *Proc. Int. Conf. Parallel Process.*, pp. II 33-40, 1992.
- [3] L.-F. Chao and E. H.-M. Sha. Scheduling data-flow graphs via retiming and unfolding. *IEEE Trans. Parallel & Distributed Syst.*, 8:1259-1267, 1997.
- [4] S. Kung, J. Whitehouse, and T. Kailath. *VLSI and Modern Signal Processing*. Prentice Hall, 1985.
- [5] C. Leiserson and J. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5-35, 1991.
- [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 & Distributed Computing & Syst.*, vol. 10, pp. 764-769, 1999.
- [7] T. O'Neil and E. H.-M. Sha. Optimal graph transformation using extended retiming with minimal unfolding. In *Proc. IASTED 12th Int. Conf. Parallel & Distributed Computing & Syst.*, vol. I, pp. 128-133, 2000.
- [8] 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 12th Great Lakes Symp. VLSI*, pp. 136-141, 2002.
- [9] T. O'Neil, S. Tongshima, and E. H.-M. Sha. Extended retiming: Optimal retiming via a graph-theoretical approach. In *Proc. ICASSP-99*, vol. 4, pp. 2001-2004, 1999.
- [10] T. O'Neil, S. Tongshima, and E. H.-M. Sha. Optimal scheduling of data-flow graphs using extended retiming. In *Proc. ISCA 12th Int. Conf. Parallel & Distributed Computing Syst.*, pp. 292-297, 1999.
- [11] K. Parhi and D. Messerschmitt. Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding. *IEEE Trans. on Comput.*, 40:178-195, 1991.