Proceedings of the 17th IASTED International Conference
Parallel and Distributed Computing and Systems
November 14-16, 2005, Phoenix, AZ, USA

STATIC SCHEDULING OF SPLIT-NODE DATA-FLOW GRAPHS

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

Edwin H.-M. Sha
Computer Science Dept.
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, 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 construct scheduling algorithms for such graphs and demonstrate our methods on specific examples.

KEY WORDS

Parallel and Distributed Compilers, Task Scheduling.

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 represented as a *data-flow graph* (*DFG*) [1]. The nodes of a DFG depict tasks, while edges between nodes symbolize 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 (DSP) [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. The effectiveness of extended retiming was further demonstated via experimentation in [6]. In all cases explored, we were able to achieve better results by using extended retiming, getting an optimal clock period while requiring less unfolding.

While the usefulness of this new transformation is clear, the result of extended retiming is a graph containing split nodes. 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 rate-optimal schedule with loop pipelining. The split-node graph is simply the most compact means for expressing this best schedule. This combination of reduced size and extensive implanted system information potentially makes the split-node DFG an attractive archetype for research into DSP on parallel embedded systems, where concision is key. However, before we get to that point, much basic work remains to be completed.

The properties of split-node graphs and the means by which they can be manipulated in order to draw out the pipelined schedule represented therein have not been explored in the literature. By using a split-node DFG to characterize 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. Thus, while many scheduling algorithms for traditional data-flow graphs exist throughout the literature [9–13], we must make specific modifications to existing methods so that they apply to this new model and produce an optimal schedule which obeys the additional rules regarding pipelining that the split-node graph dictates.

In this paper, we develop the first method designed specifically to efficiently schedule the system represented by a split-node graph. To that end, we formally define a split-node data-flow graph and redefine the terminology of scheduling to fit this new paradigm. We develop scheduling algorithms for split-node graphs, and explain how to achieve a rate-optimal schedule by applying these algorithms. Finally, we demonstrate our methods on specific examples.

2 Background

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

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; $d : E \to \mathbf{Z}$ is a function with $d(e)$ the delay count for edge e; and $t : V \to \mathbf{Z}^{\delta}$ 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 1, $t(A) = (1, 4, 3, 2)$ 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 $T(B) = T(C) = 2$.

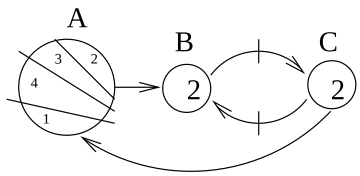


Figure 1. A sample SDG.

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 1 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 1 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. We see that it is straight-forward to derive a schedule from a simple SDG purely through observation. Our purpose is to formalize this method so that it may be applied automatically to more complex examples.

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 further define $d^+(u \to v)$ as $d(e)$ plus the number of delays within the source node u. Referring to Figure 1, we observe that $d^+(A \to B) = 3$ while $d(e) = 0$ for $e = (A, B)$.

An *integral time schedule* or *integral schedule* is a function $s : V \times \mathbf{N} \to \mathbf{Z}$ where the starting time of node v in the i^{th} iteration is given by $s(v, i)$. It is a *legal schedule* if $s(u, i) + T(u) \leq s(v, i + d^+(u \to v))$ 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.

As we've stated, 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* [14] 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: the outer $A \to B \to C \to A$ loop with total computation time 14 and delay count 4; and the $B \to C \to B$ 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. When the iteration period of the schedule equals the iteration bound of the DFG, 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 $B(G) \leq \frac{c}{f}$.

3 SDG Scheduling Algorithms

We now discuss a method for scheduling a split-node graph, based on the as-early-as-possible (AEAP) scheduling algorithm [9]. 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, a dummy source node and edges from it to all other nodes are added. The procedure for constructing this graph appears as Algorithm 1, with the sequencing graph for Figure 1 given as Figure 2(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 [15] 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, 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 2.

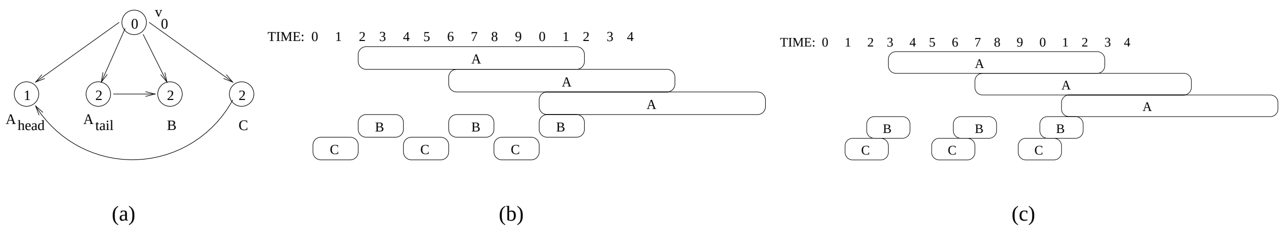


Figure 2. (a) The sequencing graph for Figure 1; (b) The forward schedule for this graph; (c) The backward schedule.

Algorithm 1 Building the sequencing graph for a split-node DFG

Input: A split-node DFG $G = \langle V, E, d, t \rangle$
Output: An acyclic sequencing graph $G' = \langle V', E' \rangle$
$V' \leftarrow \emptyset$
$E' \leftarrow \emptyset$
for all $v \in V$ **do**
  if $t(v)$ is an n-tuple **with** $n > 1$ **then**
    /* Replace all split nodes by a head and tail. */
    $V' \leftarrow V' \bigcup \{v_h, v_t\}$
    $t'(v_h) \leftarrow$first element of $t(v)$
    $t'(v_t) \leftarrow n^{th}$ element of $t(v)$
  else
    /* Any non-split node is retained as-is. */
    $V' \leftarrow V' \bigcup \{v\}$
    $t'(v) \leftarrow t(v)$
  end if
end for
for all $e = (u, v) \in E$ **with** $d(e) = 0$ **do**
  /* Edges from a split node now leaving tail. */
  if u is a split node **then**
    $\psi \leftarrow u_t$
  else
    $\psi \leftarrow u$
  end if
  /* Edges into a split node routed into head. */
  if v is a split node **then**
    $\omega \leftarrow v_h$
  else
    $\omega \leftarrow v$
  end if
  $E' \leftarrow E' \bigcup \{(\psi, \omega)\}$
end for
/* Add dummy source node. */
$V' \leftarrow V' \bigcup \{v_0\}$
$t'(v_0) \leftarrow 0$
for all $v \in V'$ **with** $v \neq v_0$ **do**
  /* Add edges from source to all other nodes. */
  $E' \leftarrow E' \bigcup \{(v_0, v)\}$
end for

Algorithm 2 Forward scheduling

Input: A split-node DFG $G = \langle V, E, d, t \rangle$ with clock period c
Output: An forward repeating schedule S
/* Apply Algorithm 1 to G */
$G' \leftarrow SequencingGraph(G)$
for all $v \in V'$ **do**
  $time(v) \leftarrow -\infty$
end for
$time(v_0) \leftarrow 0$
/* Find longest paths to all nodes in seq. graph. */
Topologically sort the vertices of V'
for all $u \in V'$ taken in sorted order **do**
  for all $v \in V'$ adjacent to u **do**
    if $time(v) < time(u) + t'(u)$ **then**
      $time(v) \leftarrow time(u) + t'(u)$
    end if
  end for
end for
for all $v \in V$ **do**
  if v is a split node **then**
    /* Schedule only the heads of split nodes. */
    $S(v, 0) \leftarrow time(v_h)$
  else
    /* Schedule the complete node if not split. */
    $S(v, 0) \leftarrow time(v)$
  end if
end for
for all $v \in V$ **and** integers $i \geq 1$ **do**
  /* Repeat to derive complete schedule. */
  $S(v, i) \leftarrow S(v, 0) + c \cdot i$
end for

As an example, return to the sequencing graph in Figure 2(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 2(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 2(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 2(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 2(c).

4 The Clock Period of an SDG

One detail passed over in Algorithm 2 is the specification of a *clock period* for our SDG. While we could use almost anything sufficiently large for an input parameter and a legal schedule would still result, we are interested in minimizing this so as to produce the best possible schedule. To this end, the minimum clock period is formally defined as the length of the longest zero-delay path (i.e. connected sequence of nodes and edges) in a graph. Informally, the clock period represents the "maximum amount of propagation delay through which any signal must ripple between clock ticks" [2].

In an SDG, "paths" are either internal pieces of split nodes, or zero-delay edge sequences from the tail of a split node through unsplit nodes into the head of another split node. We thus calculate the clock period of an SDG in two phases. First, we determine the computation time of the biggest piece of any split node. (Of course, in the case of unsplit nodes, this is the total computation time.) Next, we use the sequencing graph to find sums of computation times along paths from split-node tails to split-node heads. The maximum over this combined data set is the minimum clock period. A formalized method for this, based on the method from [2] for traditional DFGs, appears as Algorithm 3.

Algorithm 3 Determining the clock period of a split-node DFG

Input: A split-node DFG $G = \langle V, E, d, t\rangle$
Output: Its clock period c
  /* Compute zero-delay paths. */
  $G' \leftarrow SequencingGraph(G)$
  Topologically sort the vertices of V'
  $\delta(v_0) \leftarrow 0$
  for all $v \in V'$ taken in sorted order **with** $v \neq v_0$ **do**
    $\delta(v) \leftarrow t'(v) + \max\{\delta(u) : (u, v) \in E'\}$
  end for
  for all $v \in V$ **do**
    if $t(v)$ is an n-tuple **with** $n > 1$ **then**
      /* Find time of split node's largest piece. */
      $\pi(v) \leftarrow \max\{i^{th}$ element of $t(v) : i = 1, ..., n\}$
      $\Delta(v) \leftarrow \max\{\pi(v), \delta(v_h)\}$
    else
      $\Delta(v) \leftarrow \delta(v)$
    end if
  end for
  $c \leftarrow \max\{\Delta(v) : v \in V\}$

Reconsider the graph in Figure 1. One topological sort of the nodes in its sequencing graph (Figure 2(a)) yields the order v_0, C, A_{head}, A_{tail} and B. The longest paths into each of these nodes was noted above. We now compute the δ-values by adding the lengths of the paths into these nodes to the computation times of the nodes themselves (since any signal must pass through the end node as well). Thus the δ-values for A_{head}, A_{tail}, B and C are 3, 2, 4 and 2, respectively. A is the only split node and has a computation time of 4 for its largest piece. The longest zero-delay path into A is the edge from C into its head, which has length 3 as noted. Thus $\Delta(A) = 4$, the Δ-values equal the δ-values for the other nodes, and the maximum is 4, the clock period we have been using all along.

5 Analysis

The key thing to realize in constructing the sequencing graph in Algorithm 1 is that any node in the original SDG is replaced by either one or two new nodes in the sequencing graph, depending on whether or not the original node is split. Thus the first and last loops execute in $O(|V|)$ time steps, while the loop in between them at worst requires $O(|E|)$. It is therefore clear that Algorithm 1 is of complexity $O(|V| + |E|)$.

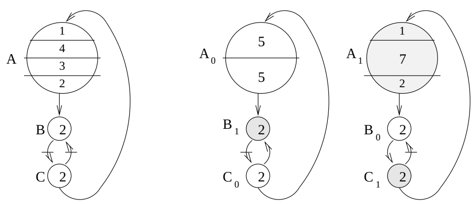
Similarly the time complexities of Algorithms 2 and 3 are $O(|V| + |E|)$ due to the topological sorts. The loop immediately following the sort in Algorithm 2 requires at worst $O(|E|)$ time steps since the SDG is a directed graph, while the remaining loops take $O(|V|)$ time. The first loop in Algorithm 3 has time complexity at worst $O(|V| + |E|)$, while the second takes $O(|V|)$ time to run.

We can therefore conclude that the time complexity of our overall scheduling process for a split-node graph is at worst $O(|V| + |E|)$. Not only have we constructed an initial method for scheduling a SDG, we have constructed one that is no more complex than similar existing methods for traditional DFGs [9, 10].

6 Achieving Rate Optimality via Unfolding

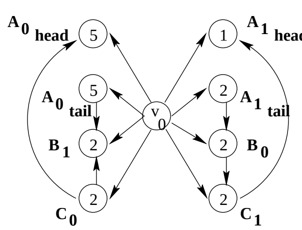
While our scheduling algorithm appears effective, this first example demonstrates its weakness. Because our algorithms require an integral clock period, the best we could accomplish was to create a schedule for Figure 1 with a near-optimal period of 4. We previously determined that the iteration bound for Figure 1 is the smaller $\frac{7}{2}$. Indeed, the fact that our AEAP and ALAP schedules differ noticably tells us that there is room for improvement. The question is then what to do with a fractional iteration period.

Unfolding [16] transforms a data-flow graph by scheduling multiple iterations simultaneously. For our particular example, if we schedule two iterations together during 7 clock cycles, we would achieve an average iteration period equal to our lower bound. In other words, if we unfold Figure 3(a) twice and then schedule with a clock period of 7, we would achieve rate-optimality. In [17], we presented an unfolding algorithm for the split-node model. Applying this algorithm to unfold our initial example twice yields the graph in Figure 3(b). For clarity, the nodes comprising iteration one are shaded, those in iteration zero are not. With unfolding complete, we now attempt to schedule the unfolded graph with a clock period of 7 time units. Algorithm 1 produces the sequencing graph in Figure 4(a). From the information contained in this graph, we can derive the time schedule for the zeroth iteration seen in Figure 4(b) below via Algorithm 2. Note that this iteration is the zeroth iteration for the *unfolded* graph and contains iterations zero and one of the original graph.



(a) (b)

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



(a)

v	$S(v,0)$	$T(v)$
A_0	2	10
A_1	6	10
B_0	2	2
B_1	5	2
C_0	0	2
C_1	4	2

(b)

Figure 4. (a) Sequencing graph for Figure 3(b); (b) The time schedule for iteration zero.

Finally, we can use the information from this table to construct a first schedule. Now, as in [18], we can optimize this schedule to arrive at the final rate- and processor-optimal schedule for Figure 1 pictured in Figure 5.

7 Example

We now review our methods by applying them to an additional example, the graph pictured in Figure 6(a). Applying

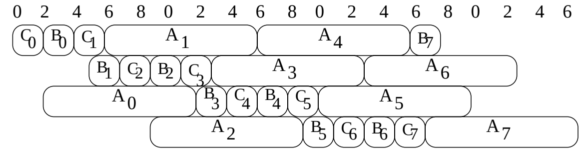


Figure 5. Rate- and processor-optimal schedule for Figure 1.

Algorithm 1 to this split-node graph produces the scheduling graph in Figure 6(b). The longest length of any path in this graph is 3, so we adopt this as our clock period. Furthermore, the longest path from v_0 to A has zero length, the longest paths to B_{head} and E_{head} each have length 1, and the longest paths to C_{head} and D each have length 2. We can thus produce the time schedule for the zeroth iteration seen in Figure 6(c) below via Algorithm 2. Propagating these values across time and optimizing for best processor assignment as in [18] yields the final time- and processor-optimal schedule in Figure 7.

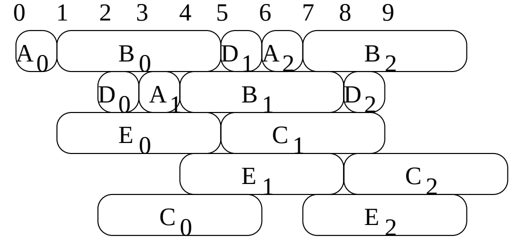


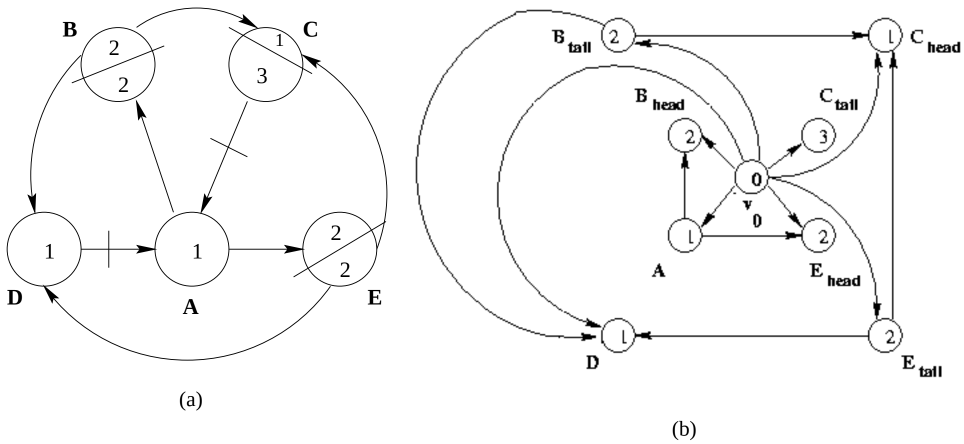
Figure 7. Final schedule for Figure 6(a).

8 Conclusion

In this paper, we have formally defined a split-node dataflow graph and redefined the terminology of scheduling to fit this new paradigm. We have developed scheduling algorithms for split-node graphs, and demonstrated how to achieve a rate-optimal schedule by applying these algorithms. Finally, we have demonstrated our methods on specific examples.

Acknowledgement

This work was partially supported by NSF grants MIP-9501006 and MIP-9704276; and 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, NSF grants ETA-0103709 and CCR-0309461, Texas ARP grant 009741-0028-2001 and the TI University program.



v	$S(v,0)$	$T(v)$
A	0	1
B	1	4
C	2	4
D	2	1
E	1	4

(c)

Figure 6. (a) Another sample SDG; (b) Its sequencing graph; (c) The time schedule for iteration zero.

References

[1] 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.

[2] C.E. Leiserson and J.B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, 1991.

[3] S.Y. 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 dataflow graphs. In *Proc. Int. Conf. Parallel Process.*, pages II 33–40, 1992.

[5] T.W. O'Neil, S. Tongsima, and E. H.-M. Sha. Extended retiming: Optimal retiming via a graph-theoretical approach. In *Proc. ICASSP-99*, volume 4, pages 2001–2004, 1999.

[6] T.W. 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.*, volume 10, pages 764–769, 1999.

[7] T.W. 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 & Distributed Computing Syst.*, pages 292–297, 1999.

[8] T.W. 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.*, volume I, pages 128–133, 2000.

[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 Process.*, 10:207–223, 1995.

[11] F. Gasperoni and U. Schwiegelshohn. Generating close to optimum loop schedules on parallel processors. *Parallel Process. Letters*, 4:391–403, 1994.

[12] F. Gasperoni and U. Schwiegelshohn. Transforming cyclic scheduling problems into acyclic ones. In *Scheduling Theory and Its Applications*, pages 241–258. John Wiley & Sons, 1995.

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

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

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

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

[17] T.W. O'Neil and E. H.-M. Sha. Unfolding a split-node dataflow graph. In *Proc. IASTED 14th Int. Conf. Parallel & Distributed Computing & Syst.*, pages 717–722, 2002.

[18] T.W. 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 Symposium on VLSI*, pages 136–141, 2002.