

Optimal Graph Transformation using Extended Retiming with Minimal Unfolding*

Timothy W. O'Neil
Dept. of Comp. Science & Eng.
University of Notre Dame
Notre Dame, IN 46556
Phone: (219) 631-8720
Fax: (219) 631-9260
Email: toneil@cse.nd.edu

Edwin H.-M. Sha
Dept. of Computer Science
Erik Jonsson School of Eng. and C.S.
Box 830688, MS EC 31
University of Texas at Dallas
Richardson, TX 75083-0688
Email: edsha@utdallas.edu

Abstract

Many computation-intensive iterative or recursive applications commonly found in digital signal processing and image processing applications can be represented by *data-flow graphs* (DFGs). The execution of all tasks of a DFG is called an *iteration*, with the average computation time of an iteration the *iteration period*. A great deal of research has been done attempting to optimize such applications by applying various graph transformation techniques to the DFG in order to minimize this iteration period. Two of the most popular are *retiming* and *unfolding*, which can be performed in tandem to achieve an optimal iteration period. However, the result is a transformed graph which is much larger than the original DFG. In our previous work, we proposed a new technique, *extended retiming*, which can be combined with minimal unfolding to transform a DFG into one whose iteration period matches that of the optimal schedule under a pipelined design. In this paper, we augment our previous work by designing an efficient retiming algorithm which may be applied directly to a DFG instead of the larger unfolded graph.

Index terms: Task Scheduling, Data-flow graphs, Retiming, Unfolding, Graph Transformation, Timing Optimization

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) [4]. 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 [8], digital signal processing [7] and program descriptions [2].

In our previous work [10–12], we proposed an efficient algorithm, *extended retiming*, which transforms a DFG into an equivalent graph with maximum parallelization. However, there remained applications for which our original framework will not deliver the best possible result. We have corrected this exclusion and demonstrated that extended retiming is a technique which, when combined with unfolding by the minimum rate-optimal unfolding factor, transforms a graph into one whose iteration period matches that of the rate-optimal schedule under a pipelined design. To the best of our knowledge, this is the first method that can do this. However, we have not yet developed an efficient method for find-

ing an extended retiming under these revised rules. We will accomplish this task in this paper.

The execution of all tasks of a DFG is called an *iteration*. A very popular strategy for maximizing parallelism is to transform the original graph by scheduling multiple iterations simultaneously, a technique known as *unfolding* [13]. While the graph becomes much larger, the average computation time of an iteration (the *iteration period*) can be reduced. In our previous work, we demonstrated that extended retiming allows us to achieve an optimal iteration period when the iteration period is an integer. We later refined our original scheme so that extended retiming may be combined with unfolding. We then showed that this combination achieves optimality in all cases. In fact, we have shown that this combination attains an optimal result while doing a minimal amount of unfolding. We find that the combination of traditional retiming and unfolding does not correctly characterize the implementation using a pipelined design and, therefore, tends to give a large unfolding factor. Thus we not only maximize parallelism by using extended retiming, but we also minimize the size of the necessary transformed graph.

In addition to unfolding, one of the more effective graph transformation techniques is *retiming*, where delays are redistributed among the edges so that the function of the DFG G remains the same, but the length of the longest zero-delay path (the *clock period* of G , denoted $cl(G)$) is decreased. This technique was introduced in [8] to optimize the throughput of synchronous circuits, and has since been used extensively in such diverse areas as software pipelining [15] and hardware-software codesign [5]. We have shown previously that neither unfolding [9] nor this traditional form of retiming [11] can produce optimal results when applied individually, but the combination will achieve optimality [4].

To illustrate these ideas, consider the example of Figure 1(a). The numbers inside the nodes represent computation times. The short bar-lines cutting the edge from node C to node B (hereafter referred to by the ordered pair (C, B)) represent inter-iteration dependencies between these nodes. In other words, the two lines cutting (C, B) tell us that task B of our current iteration depends on data produced by task C two iterations ago. This representation of such a dependency is called a *delay* on the edge of the DFG.

It is clear that the clock period of this graph is 14, obtained from the path from A to C . Since an iteration of the DFG may be scheduled within 14 time units as in Figure 1(b), the iteration period of this graph is also 14. However, if we were to remove a delay from (C, A) and place it on (A, B) , the iteration period would be reduced to 10 while not affecting the function of the graph. The

*This work is partially supported by NSF grants MIP95-01006 and MIP97-04276, and by the A. J. Schmitt Foundation.

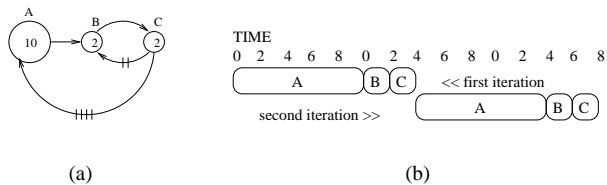


Figure 1: (a) A data-flow graph; (b) The schedule for the DAG part of Figure 1(a).

example shows how retiming may be used to adjust the iteration period of a DFG.

How small can we make our iteration period? Since retiming preserves the number of delays in a cycle, the ratio of a cycle’s total computation time to its delay count remains fixed regardless of retiming. The maximum of all such ratios, called the *iteration bound*, acts as a lower bound on the iteration period. In the case of Figure 1(a), there are only two cycles, the small one between nodes *B* and *C* with time-to-delay ratio $\frac{4}{2} = 2$, and the large one involving all nodes with ratio $\frac{14}{4}$. Thus the iteration bound for the graph is $\frac{7}{2}$.

Since the computation times of all nodes are integral, it seems impossible to get a fractional iteration period. However, recall that the iteration period is the *average* time to complete an iteration. If we can complete two iterations of our graph in 7 time units, the average will equal our lower bound, and our graph will be rate-optimal. To get these iterations together in our graph, we must unfold the graph. If we can unfold our graph *f* times to achieve this lower bound, our schedule is said to be *rate-optimal*, and *f* is called a *rate-optimal unfolding factor*. This paper shows that the minimum rate-optimal unfolding factor for a data-flow graph is the denominator of the irreducible form of the graph’s iteration bound.

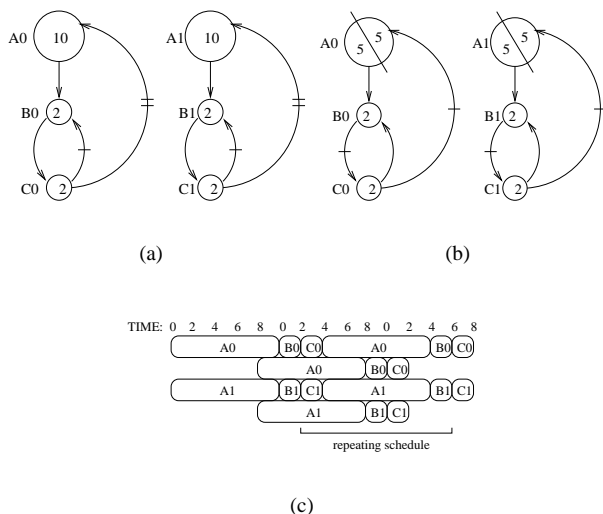


Figure 2: (a) The DFG of Figure 1(a) unfolded by a factor of 2; (b) Figure 2(a) retimed by extended retiming to be rate-optimal; (c) The optimal schedule for the retimed graph.

As an example, let’s unfold the graph of Figure 1(a) by its minimal rate-optimal factor of 2, as shown in Figure 2(a). We can schedule an iteration of this new graph—which is equivalent to

scheduling two iterations of our original graph—in the same 14 time units. We have doubled the size of our graph, but we have also reduced our iteration period to 7. We can now retime this unfolded graph as we did above to reduce our clock period to 10, which further reduces the iteration period to 5. Unfortunately this is the best we can do by unfolding twice and using traditional retiming.

If we were permitted to move a delay inside of *A*, as shown in Figure 2(b), our clock period would become 7, the iteration period would become $\frac{7}{2}$ and we would have optimized our graph, as we can see by the schedule in Figure 2(c). This is the advantage of extended retiming over traditional retiming: we are allowed to move delays not only from edge to edge, but from edge to vertex. We see from this that the combination of traditional retiming and unfolding does not completely give the correct representation of the graph’s schedule, especially when we assume a pipelined implementation.

An unfolding of 2 combined with extended retiming optimizes the graph of Figure 1(a). If we limit ourselves to traditional retiming, we must unfold the original DFG four times. After we retime in addition to unfolding, we can now schedule 4 iterations of the original graph in 14 time steps, reducing our iteration period without retiming to $\frac{7}{2}$. We see that traditional retiming tends to overestimate the rate-optimal unfolding factor, resulting in a graph that requires more resources for execution.

Note that the graph optimized by extended retiming and unfolding is half the size of that optimized by traditional retiming and unfolding. There is a very clear advantage in using extended retiming, but there have been two drawbacks to this method. First, as originally proposed, extended retiming only permitted the placement of a single delay inside any node. This was too severe a limitation for what we wanted to do, and we were able to generalize our method and deal with this problem in [10]. However, the only method for applying extended retiming and unfolding is the one we’ve outlined: unfold the graph and then retime. Since retiming is much more expensive than unfolding in terms of computation time, it is preferable to first apply extended retiming to the smaller original graph, then unfolding. We know from our results in [10] that the two operations can be applied in any order, so it makes sense that we should be able to develop such an algorithm.

In this paper, we will review our new form of retiming, *extended retiming*, which achieves an optimal result while requiring the use of a smaller unfolded graph, and thus fewer resources. It is defined in such a manner as to allow us to combine extended retiming with unfolding. When we wish to apply unfolding and extended retiming to a graph, we have two options: first retime the graph then unfold it, or unfold it then retime the unfolded graph. We have shown that these two methods are equivalent in [10]. Because of this equivalence, we are now able to design an efficient extended retiming algorithm which can be applied directly to the original graph. We show this for graphs whose iteration bounds are one or larger, which encompasses all non-trivial examples. This result improves our previous work, the application of which could produce a retiming function only for the larger unfolded graph. Finally, we will demonstrate that the minimum rate-optimal unfolding factor for a data-flow graph is the denominator of the irreducible form of the graph’s iteration bound.

2 Background

In this section, we wish to review previously presented definitions and results. We will rely on this background material [1,13]

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.

Now, let f be 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. In general, 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.

A classic result from [8] characterized the upper bound of a graph's cycle period in terms of the computation time of its longest zero-delay path. The analogous result for an unfolded graph, which we will need for our coming work, is proven in [4].

Thm 2.1 *Let G be a DFG, c a potential cycle period and f an unfolding factor.*

1. $cl(G_f) = \max\{T(p) : p \in G \text{ is a path with } D(p) < f\}$.
2. $cl(G_f) \leq c$ iff $D(p) \geq f \forall$ paths $p \in G$ with $T(p) > c$.

2.2 Extended Retiming

As in [10], we define 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 can't have a delay at this position. Also for simplicity we will express the f -tuple as $\frac{1}{t(v)}(r_1, r_2, \dots, r_f)$ or as a single fraction $\frac{r_1}{t(v)}$ when $f = 1$.

We can see from this definition that $r(v)$ can be viewed as consisting of an integer part and a fractional part. We will use the notation $\iota_r(v)$ to denote the value of this integer part, while $\mathfrak{R}_r(v)$ will be the number of non-zero coordinates in the f -tuple. We will also assume throughout this paper that the elements of an f -tuple are listed in increasing order.

As with standard retiming, we will denote the DFG retimed by r as $G_r = \langle V, E, d_r, t \rangle$. When we define the delay count of the edge $e = (u, v)$ after retiming, we must remember to include delays within each end-node as well as delays along the edge itself. Furthermore, we previously defined a path p to be a connected sequence of nodes and edges, with $D(p)$ being the path's total delay count. If we now require $D(p)$ to count the delays both among the nodes and along the edges of p , we can easily obtain these properties:

Lem 2.1 *Let G be a DFG without split nodes and r an extended retiming. Then the retimed delay count on:*

1. edge $e = (u, v)$ is $d_r(u \rightarrow v) = d(e) + \iota_r(u) - \iota_r(v) + \mathfrak{R}_r(u)$.
2. path $p : u \Rightarrow v$ is $D_r(u \Rightarrow v) = D(p) + \iota_r(u) - \iota_r(v) + \mathfrak{R}_r(u)$.
3. cycle $\ell \in G$ is $D_r(\ell) = D(\ell)$.

Given an edge $e = (u, v)$, we use $d_r(u \rightarrow v)$ 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 $d_r(e)$ as in the traditional case. As with traditional retiming, an extended retiming is *legal* if $d_r(e) \geq 0$ for all edges $e \in E$ and *normalized* if $\min_v \iota_r(v) = 0$. Note that any extended retiming can be normalized by subtracting $\min_v \iota_r(v)$ from all values $\iota_r(v)$.

2.3 Static Scheduling

Given a DFG G , 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 . Figure 3(b) shows the scheduling graph of the example in Figure 3(a) when $c = 3$ and $f = 1$. It can be shown that, if $\frac{c}{f}$ is a feasible iteration period, then the scheduling graph contains no negative-weight cycles. Define $sh(v)$ for every node v to be the length of the shortest path from v_0 to v in this modified G^s . For example, in the graph of Figure 3(b), we note that $sh(A) = 0$ and $sh(B) = sh(C) = -\frac{1}{3}$. It takes $O(|V||E|)$ time to compute $sh(v)$ for every node v [3].

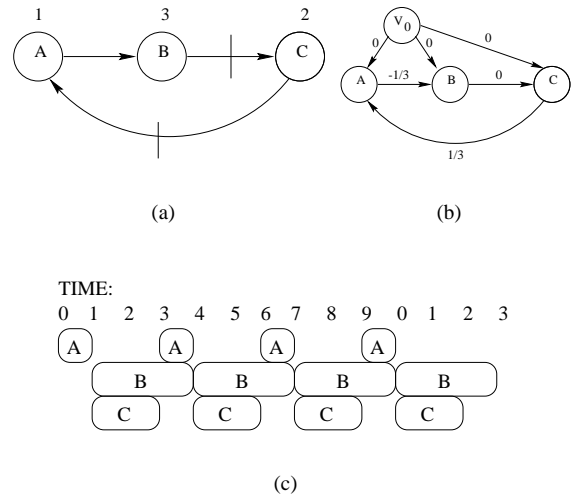


Figure 3: (a) A DFG; (b) The scheduling graph with $c = 3$ and $f = 1$; (c) The schedule with cycle period 3.

As we've stated, the iteration period is bounded from below by the *iteration bound* [14] of G , which is denoted $B(G)$ and defined to be the maximum time-to-delay ratio of all cycles in G . For example, the graph in Figure 3(a) contains only one loop, which has two delays and a total computation time of 6; thus $B(G) = 3$ for this graph. The schedule for this graph displayed in Figure 3(c)

has an iteration period of 3. In this situation, when the iteration period of a static schedule equals the iteration bound of the DFG, we say that the schedule is *rate-optimal*. The relationship between the iteration bound of a DFG and the DFG's scheduling graph is given in Lemma 3.1 of [3]:

Lem 2.2 *Let G be a DFG, c a clock period and f an unfolding factor. $B(G) \leq \frac{c}{f}$ if and only if the scheduling graph G^s contains no cycles having negative weight.*

We formally define an *integral schedule* on a DFG G to be a function $s : V \times \mathbf{N} \rightarrow \mathbf{Z}$ where the starting time of node v in the i^{th} iteration ($i \geq 0$) is given by $s(v, i)$. It is a *legal schedule* if $s(u, i) + t(u) \leq s(v, i + d(e))$ for all edges $e = (u, v)$ and iterations i . For example, the legal integral schedule of Figure 3(c) is $S_3(v, i) = 3(i - sh(v))$ for all nodes v and iterations i , where the values for $sh(v)$ are derived from the graph of Figure 3(b).

A legal schedule is a *repeating schedule* for cycle period c if $s(v, i + 1) = s(v, i) + c$ for all nodes v and iterations i . It's easy to see that $S_3(v, i)$ is an example of a repeating schedule. A repeating schedule can be represented by its first iteration, 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. If an operation of the partial schedule is assigned to the same processor in each occurrence of the partial schedule, we say that our schedule is *static*.

Since the iteration bound for the graph of Figure 3(a) is 3, and all nodes of this graph are 3 or smaller, Theorems 2.3 and 3.5 of [3] tell us that the minimum achievable cycle period for this graph is 3. We can then produce the static DFG schedule in Figure 3(c) by constructing the scheduling graph and then computing $sh(v)$ for each of the nodes. We can then use this information to create the schedule of Figure 3(c) by applying the formula from the above proof to create the schedule; for this example $S_3(A, 0) = 0$ and $S_3(B, 0) = S_3(C, 0) = 3 \cdot \frac{1}{3} = 1$.

3 Finding an Extended Retiming from a Static Schedule

As we've said throughout this paper, we currently have one method for finding an extended retiming which can be combined with a non-trivial unfolding factor to achieve optimality. In this section we will develop another, more efficient algorithm. We demonstrate our methods using the graph in Figure 1(a) with a clock period of 7 and unfolding factor 2.

Our current procedure calls for us to unfold the graph twice (as in Figure 2(a)) and then schedule it with a clock period of 7. We use DFG scheduling as defined in [3], starting with the construction of the scheduling graph in Figure 4(a). We note from this graph that $sh(A0) = sh(A1) = 0$, $sh(B0) = sh(B1) = -\frac{10}{7}$ and $sh(C0) = sh(C1) = -\frac{12}{7}$. We next construct the schedule of Figure 4(b) according to the formula $S_7(v, i) = 7(i - sh(v))$. Since this is the schedule for our unfolded graph and we will do no further unfolding, we can apply the result from [12], cutting the graph immediately before the last nodes to enter the schedule (i.e., the two copies of C) and instantly reading a legal retiming with $r(A0) = r(A1) = 1\frac{5}{10}$, $r(B0) = r(B1) = 1$ and $r(C0) = r(C1) = 0$. It is this function which yields the graph in Figure 2(b) when applied to the graph in Figure 2(a).

This function is now used to construct a legal retiming on the original graph. We add the retimings for all copies of a particular node together using a special addition operator \oplus which adds the

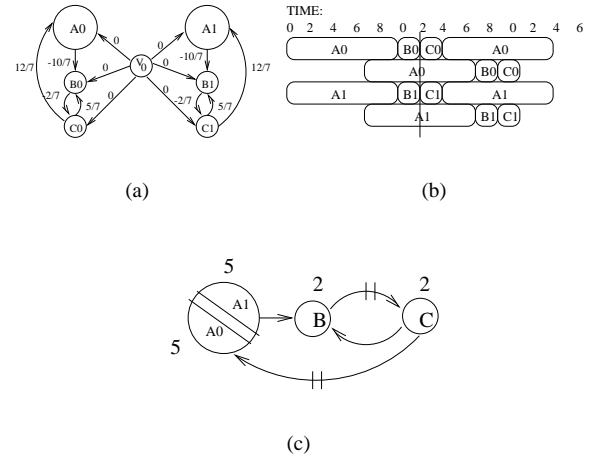


Figure 4: (a) Scheduling graph for Figure 2(a); (b) Cut schedule for this DFG; (c) The retimed DFG.

integer parts while concatenating the fractional parts. Formally, for each node u and positive integers i and j ,

$$\begin{aligned} r(u_i) \oplus r(u_j) &= \left(i_r(u_i) + \frac{1}{t(u)}(\alpha_1, \alpha_2, \dots, \alpha_n) \right) \\ &\oplus \left(i_r(u_j) + \frac{1}{t(u)}(\beta_1, \beta_2, \dots, \beta_m) \right) \\ &= (i_r(u_i) + i_r(u_j)) \\ &\quad + \frac{1}{t(u)}(\alpha_1, \alpha_2, \dots, \alpha_n, \beta_1, \beta_2, \dots, \beta_m). \end{aligned}$$

Thus, for our example, $r(A) = 1\frac{5}{10} \oplus 1\frac{5}{10} = 2 + \frac{1}{10}(5, 5)$, $r(B) = 1 \oplus 1 = 2$ and $r(C) = 0 \oplus 0 = 0$. Applying this to our original graph in Figure 1(a) results in the graph in Figure 4(c), with two delays inside of node A next to each other. The result is an optimized graph, but the process requires a great deal of time and space because we are working with the much larger unfolded graph.

In [10], we demonstrated that the order of application didn't matter; we could derive an optimal result either by unfolding then retiming or by applying retiming first. Therefore, it makes sense that we should be able to construct a method similar to the above one, but which is applied to the original graph. Let us attempt to do what we did above without the unfolding. In other words, we propose to construct our static schedule as before, based on the original graph this time. We will cut this resulting schedule and read our retiming as before.

We begin by applying this proposed algorithm to the graph in Figure 1(a). The scheduling graph with clock period 7 and unfolding factor 2 is displayed as Figure 5(a); note that $sh(A) = 0$, $sh(B) = -\frac{20}{7}$ and $sh(C) = -\frac{24}{7}$ in this case. This graph is now scheduled according to the formula $S_{7/2}(v, i) = \lceil \frac{7}{2}(i - sh(v)) \rceil$ and cut before C 's initial entrance, as in Figure 5(b). The function that we now read has $r(A) = 1 + \frac{1}{10}(1, 5, 8)$, $r(B) = 1$ and $r(C) = 0$. When applied to the original graph (as in Figure 5(c)), this function appears to be a legal retiming which does optimize the graph.

Having established what we want to do, we must formalize this method and prove its result is a legal retiming which optimizes a DFG. Let $S(v, i) = \lceil \frac{c}{f}(i - sh(v)) \rceil$ be the integral schedule with

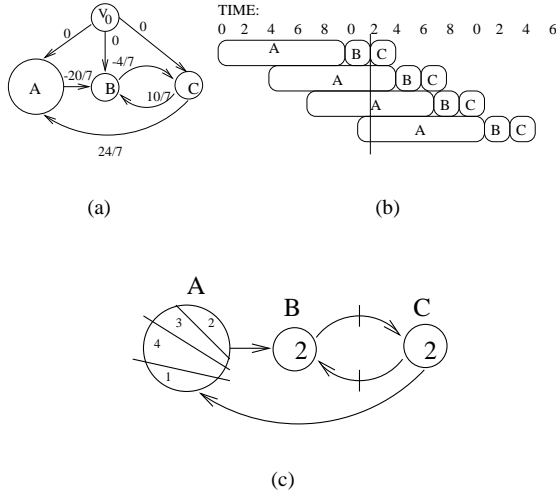


Figure 5: (a) Scheduling graph for Figure 1(a); (b) Cut schedule for this DFG; (c) The retimed DFG.

clock period c and unfolding factor f . $S(v, i)$ gives the starting time of node v in the i^{th} iteration. Thus the time at which the prologue ends, which we will denote as M and is where we want to make our cut, equals the starting time of the last node to enter the static schedule. In other words, $M = \max_v S(v, 0)$. (See that $M = S(C, 0) = 12$ above.) We now wish to count the number of either whole or partial occurrences of each node to the left of this cut. The i^{th} copy of node v begins to the left of the cut if $S(v, i) < M$. A copy of a node is complete if $M - S(v, i) \geq t(v)$; otherwise it is partial. Clearly each complete copy of a node adds 1 to the eventual retiming function. On the other hand, if a copy is cut, we only want to add the fraction of the node to the left of the cut, which is found by dividing the piece's computation time by the computation time of the whole node. At the end, we combine the contributions from a node's copies via our \oplus operator, finally arriving at the retiming formula

$$r(v) = \bigoplus_{i: S(v, i) < M} \min \left\{ 1, \frac{M - S(v, i)}{t(v)} \right\}. \quad (1)$$

Let us consider this formula when applied to node A of Figure 1(a). As we can see from our schedule in Figure 5(b), the first four iterations of A are to be considered when constructing the node's retiming:

1. $S(A, 0) = 0$ and $\min \left\{ 1, \frac{12}{10} \right\} = 1$.
2. $S(A, 1) = \lceil \frac{7}{2} \cdot 1 \rceil = 4$ and $\min \left\{ 1, \frac{12-4}{10} \right\} = \frac{8}{10}$.
3. $S(A, 2) = \lceil \frac{7}{2} \cdot 2 \rceil = 7$ and $\min \left\{ 1, \frac{12-7}{10} \right\} = \frac{5}{10}$.
4. $S(A, 3) = \lceil \frac{7}{2} \cdot 3 \rceil = 11$ and $\min \left\{ 1, \frac{12-11}{10} \right\} = \frac{1}{10}$.

Combining these figures gives us $r(A) = 1 \oplus \frac{8}{10} \oplus \frac{5}{10} \oplus \frac{1}{10} = 1 + \frac{1}{10}(1, 5, 8)$, exactly the same answer we found by simply examining the schedule table. Our formula appears to accurately describe this situation.

We must show that (1) is, in fact, a legal extended retiming which minimizes the iteration period of a data-flow graph. Recall that these definitions are based on the ν_r and \mathfrak{R}_r functions for the retiming r in question. Therefore, before proceeding to our primary result, we must find the closed forms of these functions for our proposed formula.

Lem 3.1 Let r be the extended retiming given by Equation (1) above. Then $\nu_r(v) = \lfloor \frac{f}{c}(M - t(v)) + sh(v) \rfloor + 1$ and $\mathfrak{R}_r(v) = \lfloor \frac{f}{c}(M - 1) + sh(v) \rfloor + 1 - \nu_r(v)$.

Proof: Lemma 6.2 of [9]. \square

With this result we can now show:

Thm 3.1 Let $G = \langle V, E, d, t \rangle$ be a DFG with iteration period $\frac{c}{f} \geq 1$, i.e. with clock period c and unfolding factor f . Then the retiming r described by Equation (1) is a legal extended retiming on G such that $cl(G_{r,f}) \leq c$ if and only if the scheduling graph G^s contains no negative-weight cycle.

Proof: Theorem 6.3 of [9]. \square

4 Minimum Rate-Optimal Unfolding Factors

If the iteration period of a graph's schedule equals the graph's iteration bound, the schedule is said to be *rate-optimal*. As we've said throughout this paper, our goal is to achieve rate-optimality via retiming and unfolding. If a data-flow graph can be unfolded f times and achieve rate-optimality (i.e. a clock period equal to $f \cdot B(G)$), we say that f is the *rate-optimal unfolding factor* for G . Obviously we wish to achieve rate-optimality while unfolding as little as possible. To this end we need to compute the minimum rate-optimal unfolding factor for any graph. We begin by showing this link between a graph's clock period and iteration bound:

Lem 4.1 Let G be a data-flow graph without split nodes, c a cycle period and f an unfolding factor with $f \leq c$. Then there exists a legal extended retiming r on G such that $cl(G_{r,f}) \leq c$ if and only if $B(G) \leq \frac{c}{f}$.

Proof: Follows from Lemma 2.2 and Theorem 3.1. \square

With this in hand we can show:

Thm 4.1 Let G be a data-flow graph without split nodes whose iteration bound exceeds 1. Let ℓ be a critical cycle of G , i.e. $B(G) = \frac{T(\ell)}{D(\ell)}$. Let g be the greatest common divisor of $T(\ell)$ and $D(\ell)$. Then $\frac{D(\ell)}{g}$ is the minimum rate-optimal unfolding factor for G .

Proof: Theorem 7.2 of [9]. \square

In short, to find the rate-optimal unfolding factor of a data-flow graph G , we compute $B(G)$ (a polynomial-time operation [6]) and reduce the resulting fraction to lowest terms. The denominator of this fraction is our desired unfolding factor.

5 Experimental Results

Let's consider the data-flow graph representation of a IIR filter. Assume that a multiplier (shown below as a circle) require four units of computation time, as opposed to one for an adder (shown as a square). Furthermore, to complicate our example, multiply the register count of each edge by 2, referred to in [8] as applying a *slowdown* of 2 to our original circuit. The result is pictured in Figure 6(a).

The resulting circuit has an iteration bound of 3, and can be retimed via extended retiming to achieve this clock period as in Figure 6(b) without unfolding. However, if we restrict ourselves to traditional retiming, the best clock period we can get is 4. The only way to obtain an optimal result is to unfold the graph by a

Benchmark	Comp. Time		Slow-down	Iter. Bound	Min. Optimal Unf. Factor		Iter. Pd. w/ Bold U. F.	
	+	×			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	$\frac{11}{2}$	2	6	5.5	12.5
All-Pole Lattice Filter	2	5	12	$\frac{3}{2}$	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	$\frac{7}{2}$	2	8	3.5	6
Fifth Order Elliptic Filter	2	30	20	$\frac{11}{2}$	2	20	5.5	15

Table 1: Experimental results for common circuits

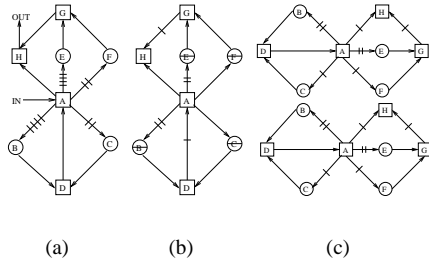


Figure 6: (a) A 2-slow second order IIR filter; (b) This graph optimized by extended retiming; (c) This graph optimized by traditional retiming

factor of 2 and retime for a clock period of 6, as shown in Figure 6(c).

Repeating this exercise with other common filters yields Table 1. 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 forces us to decide between two poor options: If we want an optimal clock period we must unfold by a larger factor, which is listed for each example in the second-to-last column of Table 1. 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). The result is a smaller circuit running at less than optimal speed.

6 Conclusion

Our original work on this topic [11,12] yielded an extended retiming method that allowed us to transform any data-flow graph to one whose clock period matched the cycle period of any of its legal schedules. We also demonstrated a simple method for finding an extended retiming which yielded a desired clock period. In our next paper [10], we improved these results by combining our method with unfolding, resulting in a more general form of extended retiming. This new result indicated to us that we should be able to find a retiming immediately without unfolding first. In this paper, we have constructed a method to do this, based on our earlier simplified algorithm. We have also derived the minimum rate-optimal unfolding factor for a data-flow graph.

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. Static scheduling for synthesis of DSP algorithms on various models. *J. VLSI Signal Process.*, 10:207–223, 1995.
- [4] 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.
- [5] K. S. Chatha and R. Vemuri. RECOD: A retiming heuristic to optimize resource and memory utilization in HW/SW codesigns. In *Proc. Codes/CASHE '98*, pp. 139–143, 1998.
- [6] A. Dasdan and R. Gupta. Faster maximum and minimum mean cycle algorithms for system-performance analysis. *IEEE Trans. CAD of Integrated Circuits & Syst.*, 17:889–899, 1998.
- [7] S. Kung, J. Whitehouse, and T. Kailath. *VLSI & Modern Signal Process.* Prentice Hall, 1985.
- [8] C. Leiserson and J. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, 1991.
- [9] T. O'Neil and E. H.-M. Sha. Combining extended retiming and unfolding for rate-optimal graph transformation. Tech. Rept. 99-12, Univ. of Notre Dame, 1999. Available online at www.cse.nd.edu/tech_reports/.
- [10] 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.
- [11] T. O'Neil, S. Tongsima, and E. H.-M. Sha. Extended retiming: Optimal retiming via a graph-theoretical approach. In *Proc. ICASSP-99*, vol. 4, pp. 2001–2004, 1999.
- [12] 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 & Distributed Computing Syst.*, pp. 292–297, 1999.
- [13] K. Parhi and D. Messerschmitt. Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding. *IEEE Trans. Comput.*, 40:178–195, 1991.
- [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] F. Sanchez and J. Cortadella. Reducing register pressure in software pipelining. *J. Inf. Sci. & Eng.*, 14:265–279, 1998.