Journal of VLSI Signal Processing 31, 207–229, 2002
© 2002 Kluwer Academic Publishers. Manufactured in The Netherlands.

Minimizing Buffer Requirements under Rate-Optimal Schedule in Regular Dataflow Networks*†

R. GOVINDARAJAN
Supercomputer Edn. & Res. Centre, Computer Science & Automation, Indian Institute of Science, Bangalore, 560 012, India

GUANG R. GAO
Electrical & Computer Engineering, University of Delaware, Newark, DE 19716, USA

PALASH DESAI
Conductus, Inc., 969 West Maude Ave., Sunnyvale, CA 94085, USA

Received May 4, 1999; Revised August 4, 2000

Abstract. Large-grain synchronous dataflow graphs or multi-rate graphs have the distinct feature that the nodes of the dataflow graph fire at different rates. Such multi-rate large-grain dataflow graphs have been widely regarded as a powerful programming model for DSP applications. In this paper we propose a method to minimize buffer storage requirement in constructing rate-optimal compile-time (MBRO) schedules for multi-rate dataflow graphs. We demonstrate that the constraints to minimize buffer storage while executing at the optimal computation rate (i.e. the maximum possible computation rate without storage constraints) can be formulated as a unified linear programming problem in our framework. A novel feature of our method is that in constructing the rate-optimal schedule, it directly minimizes the memory requirement by choosing the schedule time of nodes appropriately. Lastly, a new circular-arc interval graph coloring algorithm has been proposed to further reduce the memory requirement by allowing buffer sharing among the arcs of the multi-rate dataflow graph.

We have constructed an experimental testbed which implements our MBRO scheduling algorithm as well as (i) the widely used periodic admissible parallel schedules (also known as block schedules) proposed by Lee and Messerschmitt (*IEEE Transactions on Computers*, vol. 36, no. 1, 1987, pp. 24–35), (ii) the optimal scheduling buffer allocation (OSBA) algorithm of Ning and Gao (*Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Charleston, SC, Jan. 10–13, 1993, pp. 29–42), and (iii) the multi-rate software pipelining (MRSP) algorithm (Govindarajan and Gao, in *Proceedings of the 1993 International Conference on Application Specific Array Processors*, Venice, Italy, Oct. 25–27, 1993, pp. 77–88). Schedules generated for a number of random dataflow graphs and for a set of DSP application programs using the different scheduling methods are compared. The experimental results have demonstrated a significant improvement (10–20%) in buffer requirements for the MBRO schedules compared to the schedules generated by

*This work was done when the authors were at McGill University, Montreal, Canada, and the work was supported by research grants from Micronet—Network Centres of Excellence, Canada.
†A preliminary version of this paper has appeared in the 1994 International Conference on Application Specific Array Processors, San Franscisco, August 1994.

the other three methods, without sacrificing the computation rate. The MBRO method also gives a 20% average improvement in computation rate compared to Lee's Block scheduling method.

Keywords: buffer minimization, dataflow graphs, Digital Signal Processing (DSP) computation, Multi-Rate Software Pipelining, Regular Stream Flow Graphs, software pipelining

1. Introduction

Large-grain synchronous dataflow graphs (originally proposed by Lee and Messerschmitt [1] and subsequently used by [2, 3]) have been widely accepted as a powerful programming model for DSP applications. Here large grain means that each task in the applications generates (or consumes) multiple, sometimes in tens or even in hundreds, samples per invocation. Each of these tasks represent operations such as an FIR filter, FFT, or bandshifting. Synchronous means that the amount of input consumed by each task, and the amount of output generated, are known a priori and are fixed. Large-grain regular dataflow networks are important as they can represent a large class of DSP computations, such as speech coding, auto-correlation, power spectrum, voice band modulation, and a variety of filters [1, 2].

A large-grained dataflow graph consists of tasks (also called nodes or actors) and arcs (or channels) representing communication between tasks. The communication can be viewed as a sequence of *tokens* passing through an arc. The temporal sequences of values passed along the communication channels can be viewed as *streams*. A Stream is an abstraction of temporal sequence of values. Therefore, in this paper, we call these graphs Regular Stream Flow Graphs (RSFGs). (These graphs are called synchronous dataflow graphs by Lee and Messerschmitt [1].) Actually, the RSFG model is a subset of the more general Stream Flow Graph model developed in [4] for DSP applications.

In this paper, we investigate the construction of efficient compile-time (static) schedules as opposed to run-time (dynamic) schedules for DSP programs expressed using RSFGs. Compared to the scheduling of conventional (also known as homogeneous) dataflow graphs [5], a distinct feature of the scheduling problem for RSFGs is that the production and consumption rates of tokens on an arc in an RSFG may be different. As a consequence the actors of the RSFG may be executed at different rates—hence the name *multi-rate*—in order not to accumulate tokens on any of the arcs.

We are interested in the class of RSFGs which may contain feedback loops representing the precedence constraints between successive iterations. In a schedule for such an RSFG, operations from successive iterations can be overlapped without violating the precedence constraints. Such overlapped or *software-pipelined* schedules [6–8] are desirable as they maximize the *computation rate*, which is measured in number of iterations completed per unit time.

A major challenge facing researchers in constructing schedules for RSFGs is taking tradeoff decisions between computation rate and the storage requirements.[1] In other words,

> Given an RSFG, how does one find a schedule which minimizes buffer space requirement while still executing the program graph at an optimal computation rate?

By optimal computation rate, we mean the maximum possible computation rate without any storage constraints. Henceforth, we refer to the above problem as Minimizing Buffer requirements under Rate-Optimal schedules (MBRO) problem.

Our earlier work on scheduling RSFGs, called Multi-Rate Software Pipelining (MRSP) [9, 10], discusses a method to construct rate-optimal schedules without taking buffer constraints into account. In this work, we extend this framework to address the MBRO problem. We demonstrate that the constraints to minimize the buffer requirements under rate-optimal schedules can be formulated as a unified linear programming problem. The objective of this linear programming problem is to minimize the total buffer requirements. A novel feature of our scheduling framework is that it directly minimizes the memory requirement by choosing the schedule time of the nodes appropriately in constructing the schedule for the optimal computation rate. Lastly, to further reduce the memory requirement of our MBRO schedules, we develop a method that allows sharing of buffers among the arcs of the RSFG. We propose a new greedy algorithm for performing buffer sharing. One major application of the proposed MBRO formulation, its solution, and the greedy algorithm for buffer sharing is in software synthesis of DSP applications [3, 11, 12].

Both in constructing the MBRO schedules, and in reducing the memory requirement by buffer sharing, we are concerned, in this paper, only about the memory requirement of data (streams) communicated explicitly by the arcs of an RSFG. The storage requirement for the internal variables of the nodes of the RSFG is not considered here since this is abstracted at the RSFG model. Hence, without loss of generality, we have assumed that the storage for these internal variables do not change with different schedules. However, when the storage requirement for internal variables do change with various schedules, these variables can be modeled by explicit self arcs on the nodes.

In order to evaluate the performance of our scheduling algorithm and to compare it with other scheduling methods, we have constructed an experimental scheduling testbed. In this testbed, in addition to our MBRO scheduling method, we have implemented three other scheduling algorithms, namely (i) the periodic admissible parallel schedules, also known as block schedules, proposed by Lee and Messerschmitt [1], (ii) the optimal scheduling buffer allocation (OSBA) algorithm of Ning and Gao [13], and (iii) our own multi-rate software pipelining (MRSP) algorithm [9, 10]. As in [14], to obtain general trends over a broad range of test inputs, we propose to evaluate our scheduling method on randomly generated RSFGs. We will supplement these results with those obtained from a set of sample DSP algorithms. The buffer requirements of the schedules generated by the various scheduling algorithms are compared. The experimental results have demonstrated a significant improvement (10–20% on the average) in buffer requirements for the MBRO schedules compared to schedules generated by the other three methods. Compared to block schedules, MBRO schedules perform better in terms of both computation rate and buffer requirements. However, the execution time of the block scheduling method to construct a schedule (i.e., scheduling time) is significantly lower than the other three scheduling methods. Lastly, the proposed buffer sharing algorithm further reduces the memory requirement, on the average, by 5.2% in the MBRO schedules.

The rest of this paper is organized as follows. The following section motivates our work with the help of an example. Section 3 deals with the formulation of the Minimum Buffer Rate-Optimal (MBRO) scheduling problem. In Section 4, a solution method for the MBRO formulation is presented. In Section 5, we report experimental results, comparing MBRO formulation with other scheduling methods. In Section 6, we present our algorithm to perform buffer sharing and also report some initial experimental results. A discussion of related work is included in Section 7. Concluding remarks are presented in Section 8.

2. Motivating Example

In this section first we present the necessary background on Regular Stream Flow Graphs (RSFGs). In Section 2.2, we informally introduce the concept of software pipelining for the RSFG model. Subsequently, we will motivate our work on minimizing buffer requirements in rate-optimal schedules for multi-rate RSFGs using a simple example.

2.1. Regular Stream Flow Graphs

Regular stream flow graphs have been proposed to express DSP computations in a graphical representation. A brief introduction of RSFG is presented here, and for more information the readers are referred to [4].

An RSFG is a directed graph where nodes represent *regular* actors and directed arcs represent data communication channels or FIFO buffers. A regular actor has a fixed number of input channels and a fixed number of output arcs. Furthermore, with each arc (u, v), two fixed integers I_{uv} and O_{uv} are associated which represent respectively the number of tokens consumed by node v and the number of tokens produced by u in each firing. A node v is said to be *enabled* when it has at least I_{uv} tokens (data values) on each of its input arcs (u, v) (refer to Fig. 1). When the node fires, it produces O_{vw} tokens on each of its output arc (v, w). Formally,

Definition 2.1 (Regular Stream Flow Graph (RSFG)). An RSFG is a graph $G = [V, E, I, O, m]$ where V is a set of *regular actors*, E is a set of directed edges representing communication channels between the actors. The *Input*, *Output* and *marking* functions associated with each arc (u, v) in E represent the following:

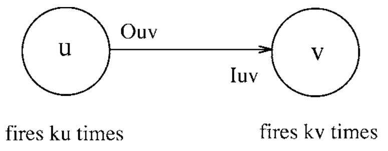


Figure 1. An arc in the RSFG.

I_{uv} represents the number of tokens (data values) consumed by v in each firing (refer to Fig. 1).
O_{uv} represents the number of tokens produced by u in each firing.
m_{uv} represents the initial number of tokens on arc (u, v).

The initial number of tokens on an arc, also known as *dependence distance*, essentially characterizes possible inter-iteration dependency between the producer and the consumer. If every actor of an RSFG is homogeneous, that is consume/produce exactly one token on each input/output arc in every firing, then the graph is a special case of RSFG and called the *homogeneous dataflow graph* [5]. Homogeneous dataflow graphs are widely used to represent programs in the dataflow model of computation.

A potential problem in RSFGs is that since the rate at which tokens are produced in a channel may be different from the rate at which they are consumed, there could be potential token accumulation. Further, since RSFGs, like synchronous data flow graphs, represent computations that are intrinsically non-terminating, token accumulation may lead to infinite memory (buffer) requirements. In a well behaved RSFG the firing rates of different actors may be different such that, effectively, there is neither token accumulation nor token depletion on any arc [4]. Intuitively, this is referred to as *consistency* [15] or *well-behavedness* [4]

The multiple firing rates of the nodes of an RSFG can be obtained by solving a set of flow balance equations, relating the number of firing of the actors of a graph and the number of tokens produced (or consumed) on each channel. More specifically, for each arc (u, v) the flow balance equation is

$$k_u * O_{uv} = k_v * I_{uv}$$

where k_u and k_v represent the firing rates of nodes u and v respectively. Solving the set of flow balance equations for all the arcs of the RSFG, we get the firing rates of the nodes. Note that the set of equations may have more than one solution. We refer to a solution as a fundamental solution if there exists at least two values k_u and k_v that are relatively prime to each other. Henceforth in this paper, the firing rates of the nodes of an RSFG always correspond to the fundamental solution.

In order to make the above discussion clear, we present an illustrative example. Consider the RSFG shown in Fig. 2. Solving the flow balance equations for this RSFG, we get the firing rates of the actors a,

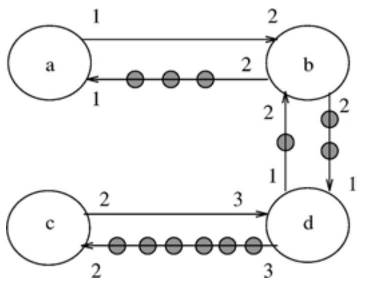


Figure 2. An example RSFG.

b, c, and d as 2, 1, 3, and 2 respectively. Let the initial number of tokens on the arcs (d, c), (b, a), (b, d), and (d, b) be six, three, two, and one respectively. It may be observed that firing the actors in the above rate restores the number of tokens on each arc to the initial value.

Informally speaking, RSFGs have the same expressive power as synchronous dataflow graphs proposed by Lee and Messerschmitt [1] which are capable of expressing a reasonably large subset of DSP application programs. We can extend RSFGs to include special actors, such as merge and switch actors, to represent program involving conditional expressions, linear recurrence, and regular iterative computation [4]. For the purpose of this paper, we focus only on RSFGs.

2.2. *Software Pipelining of Regular Stream Flow Graphs (RSFG): A Motivating Example*

Next we introduce the notion of software pipelining for multi-rate computations. In this paper we do not address issues relating to (processor) resource constraints. Further, as in [9, 10], we do not allow two instances of an actor to fire concurrently. This is because, typically in a DSP computation, such a large grain node may maintain certain state information which would be modified in each execution and used in the subsequent firing (e.g., an FIR filter). Further as will be pointed out in Section 3.1, this assumption doesn't restrict the applicability of the proposed formulation.

Let the execution time of actor a be 3 time units while that of other actors be unity. A schedule for the RSFG is shown in Table 1 where the actors (belonging to one or many iterations) in a single column fire concurrently. The symbol _a is used in Table 1 to indicate that the execution of actor a is continuing from the previous time step. The schedule of Table 1 is repetitive starting from time step 3, with a repetitive pattern from time step 3 to time step 8. A new iteration is initiated after every 6 time units, or equivalently the computation rate

Table 1. Schedule for motivating example.

	Time step												
	0	1	2	3	4	5	6	7	8	9	10	11	12
Iteration = 0	c, a	c, _a	c, _a	d, a	d, _a	_a	b						
Iteration = 1							c, a	c, _a	c, _a	d, a	d, _a	_a	b
Iteration = 2													c, a

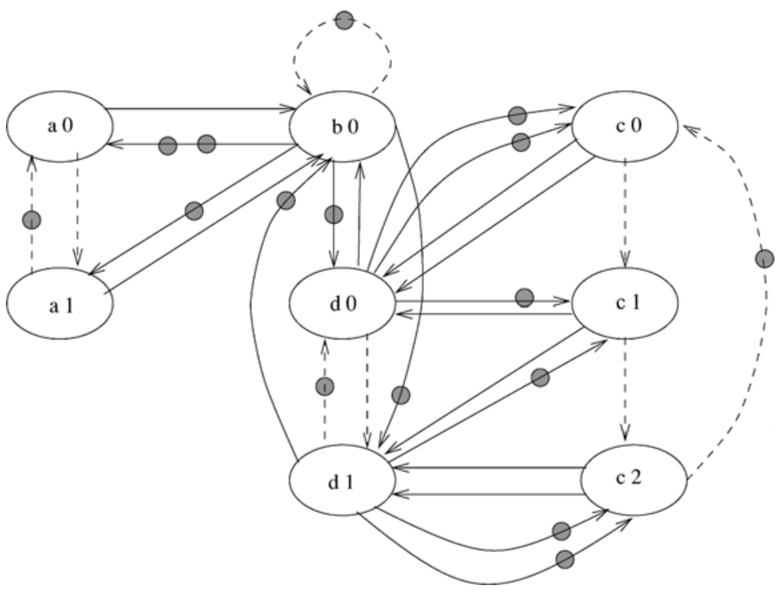


Figure 3. Equivalent homogeneous graph for Example 2.1.

is $\frac{1}{6}$. Observe further that in the repetitive pattern, the firings of the actors in various iterations overlap, and the schedule is called *software pipelined.* More details on software pipelining for RSFGs can be found in [9, 10].

The maximum computation rate of an RSFG can be derived by inspecting the equivalent *homogeneous dataflow graph* (of the original RSFG) where each actor will consume (output) only one token per arc at each firing. The equivalent homogeneous graph, shown in Fig. 3, for the motivating example consists of multiple instances of each actor (as dictated by its firing rate) and arcs indicating data dependencies between the various instances of the actors. Multiple instances of a node are represented by the actor name followed by the instance number, as in $a0$, $a1$, $c0$, $c1$, and $c2$. A method to construct the equivalent homogeneous graph is given in [10, 14].

The following theorem due to Reiter characterizes the maximum rate [16]. The maximum computation rate $\frac{1}{T}$ of a homogeneous graph is bounded by the *critical cycles* of the graph which have the smallest $\frac{M(C)}{D(C)}$ value, where $M(C)$ is the total number of tokens (or delays) on the arcs of the cycle and $D(C)$ is the sum of the execution time of the nodes in the cycle. It can be seen that the optimal rate for the graph shown in Fig. 3 is $\frac{1}{6}$, corresponding to the critical cycle, $a0 \rightarrow a1 \rightarrow a0$. Observe that this optimal computation rate equals the computation rate of the schedule given in Table 1. Since the RSFG and the corresponding homogeneous dataflow graph are equivalent [4], the maximum computation rates of the two graphs are equal. Hence the schedule shown in Table 1 is rate-optimal.

2.3. The Storage Requirements of Rate-Optimal Schedules

Next we compute the buffer requirement for each arc of the RSFG using an operational model. In this method we compute the buffer requirements for an arc by firing the nodes of the RSFG according to the schedule and by

Table 2. Buffer requirements on arcs (c, d) and (d, c).

Time step	Actors fired[a]	Buffer size on arc (c, d)		Buffer size on arc (d, c)	
		Before	After	Before	After
6	c	0	2	6	4
7	c	2	4	4	2
8	c	4	6	2	0
9	d	6	3	0	2
10	d	3	0	2	4
11	–	0	0	4	6

[a]Only actors c and d are considered here.

determining the maximum tokens present on the arc at any time during the repetitive pattern. In this paper we consider only the buffer requirements of the repetitive pattern of the schedule. Since the prologue is executed only once, separate buffer allocation can be done for it. Further, it is the buffer requirement of the repetitive pattern, rather than that of the prologue, that is likely to have significant impact on the performance of the schedule.

In computing the buffer requirements for the arcs of the RSFG, we make the following assumption without any loss of generality. Input tokens remain on the input arc until the activation (firing) is completed and output tokens are produced (all at once) at the end of the firing. This assumption is not crucial to our formulation and similar formulations can be derived for other assumptions on the time tokens are consumed (or produced). Table 2 shows the number of buffer locations used for arcs (c, d) and (d, c) after each time step in the repetitive pattern. Also, without loss of generality, we will consider the repetitive pattern starting from time step 6.

Clearly, under the given schedule, the maximum buffer size required for each of the arcs (c, d) and (d, c) is 6. A buffer size of 2 is required for the arcs (a, b) and (b, d), and a buffer size of 3 is required for (b, a) and (d, b). Thus the total buffer requirement for the schedule of Table 1 is 22.

Consider the alternative optimal rate schedule shown in Table 3 for our motivating example. The buffer requirements for the arcs (c, d) and (d, c) at each time step, starting from $t = 6$ are shown in Table 4. The maximum buffer size required for arcs (c, d) and (d, c) are 5 and 4 respectively. Hence, the total buffer requirement for all the arcs in the RSFG is 18 using the schedule shown in Table 3. In other words, by choosing the schedule of Table 3 we can reduce the buffer requirement without compromising the computation rate. Thus, the motivating example elucidates the MBRO problem introduced in Section 1. The problem is how to chose, among the rate-optimal schedules, one that has lesser memory requirement. In this paper, we answer this question by deriving linear constraints for buffer requirements and integrating them with our earlier formulation [9].

3. Minimizing Buffer Requirements in Rate-Optimal Schedules (MBRO): Problem Formulation

In [9], the problem of obtaining rate-optimal schedules for RSFGs (without buffer constraints) has been formulated as a linear programming problem, called the Multi-Rate Software Pipelining (MRSP) problem. In Section 3.1 we briefly review the MRSP formulation as well as introduce the necessary terminology for the rest of this section. Readers who are familiar with the MRSP formulation can skip this subsection. In Section 3.2, we show that the buffer storage requirements can also be represented as linear constraints, and the MRSP formulation can be extended to present a unified linear program formulation of the MBRO problem, capturing both the precedence and storage constraints.

3.1. Rate-Optimal Schedules

A schedule σ is a function which maps the ith instance or $(i + 1)$-th firing of an actor u in the RSFG to a time

Table 3. An alternative schedule for the motivating example.

	Time step													
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Iteration = 0	c, a	$\_a$	$c, \_a$	a	$d, c, \_a$	$\_a$	b	d						
Iteration = 1							c, a	$\_a$	$c, \_a$	a	$d, c, \_a$	$\_a$	b	d
Iteration = 2													c, a	$\_a$

Table 4. Buffer requirements for the alternative schedule.

Time step	Actors fired[a]	Buffer size on arc (c, d) Before	Buffer size on arc (c, d) After	Buffer size on arc (d, c) Before	Buffer size on arc (d, c) After
6	c	3	5	3	1
7	d	5	2	1	4
8	c	2	4	4	2
9	–	4	4	2	2
10	d, c	4	3	2	3
11	–	3	3	3	1

[a]Only actors c and d are considered here.

instance t. That is, $\sigma(i, u)$ represents the time t at which the i-th instance of actor u begins its execution. The instance number i is counted from 0 rather than from 1. Therefore the i-th instance of an actor actually corresponds to the $(i + 1)$-th firing of the actor.

Consider an arc (u, v) in an RSFG (refer to Fig. 1). Let O_{uv} and I_{uv} represent, respectively, the number of tokens producd by u and consumed by v in each firing. The execution times of nodes u and v are denoted by d_u and d_v respectively. Further, let $b_{uv}(t)$ denote the number of tokens present in the arc connecting u to v at time t. We will use the special notation m_{uv} to represent the initial number of tokens on the arc (u, v).

Definition 3.1. A schedule σ is said to be *admissible* if for every arc (u, v) in the graph, and for all $i \geq 0$,

$$\sigma(i, v) = t \Rightarrow b_{uv}(t) \geq I_{uv}.$$

That is, a schedule is *admissible* if it does not violate the dependency constraints imposed by the arcs of the RSFG. The following necessary and sufficient condition for the class of admissible schedules for RSFGs has been established in [9].

Theorem 3.1. *A schedule σ is admissible if and only if, for each arc (u, v) in the graph,*

$$\sigma(i, v) \geq \sigma\left(\left\lceil \frac{(i+1) * I_{uv} - m_{uv} - O_{uv}}{O_{uv}} \right\rceil, u\right) + d_u, \tag{1}$$

where d_u is the execution time of node u.

An *iteration* of an RSFG is formally defined as:

Definition 3.2. If $[k_{v_1}, k_{v_2}, \ldots, k_{v_n}]$ represents the (fundamental) firing rates of the actors of the RSFG, then an iteration of the graph consists of k_{v_i} firings of actor v_i for every $v_i \in \{v_1, v_2, \ldots, v_n\}$.

Here the term 'fundamental firing rates' refers to the fact that the rates (i.e. the $k'_{v_i}s$) do not have a common divisor. In other words, the greatest common divisor of the k_{v_i}'s is 1.

Next, we formalize the notion of r-periodic schedules for RSFGs. Intuitively, in an r-periodic schedule, the firing of the k-th instance of an actor in every r iterations takes place exactly after T time units.

Definition 3.3. A schedule is called *r-periodic* with period T if for any node v_i, for $0 \leq j < r$, and for any k in the semi-open interval $[0, k_{v_i})$ the k-th instance in iterations $j, j+r, j+2r, \ldots$, take place, respectively, at time $t_{v_i,k}, T + t_{v_i,k}, 2T + t_{v_i,k}, \ldots$

Since r iterations of the RSFG are executed in T time steps, the computation rate of an r-periodic schedule is $\frac{r}{T}$.

The notation (j, k) is used to represent the k-th instance in the j-th iteration. Similarly, $\sigma(j, k, v)$ denotes the time at which (j, k)-th firing of v commences. Finally, we use r-periodic schedules of the form $\lfloor \frac{j*T+A[k,v]}{r} \rfloor$ for the (j, k)-th firing of actor v. That is,

$$\sigma(j, k, v) = \left\lfloor \frac{j * T + A[k, v]}{r} \right\rfloor, \tag{2}$$

where T is the period of an r-periodic schedule for the given RSFG, and $\forall k \in [0, k_v)$ and $\forall v$, $A[k, v] \geq 0$ are constants that uniquely determine a schedule for the given T and r values. Now, using Eq. (2) in the necessary and sufficient condition (Inequality (1)), and after a few algebraic manipulations, we obtain the following constraints on $A[k, v]$'s (refer to [9] for the details).

$\forall(u, v)$ in the RSFG, and $\forall k \in [0, k_v)$,

$$\begin{aligned} &A[k, v] \\ &\quad - A\left[\left(\left\lceil \frac{(k+1) * I_{uv} - m_{uv} - O_{uv}}{O_{uv}} \right\rceil \bmod k_u\right), u\right] \\ &\geq T * \left\lfloor \frac{1}{k_u} \left\lceil \frac{(k+1) * I_{uv} - m_{uv} - O_{uv}}{O_{uv}} \right\rceil \right\rfloor + r * d_u \end{aligned} \tag{3}$$

Our assumption that two instances of an actor are not allowed to fire concurrently can be enforced by including a self arc to each actor in the RSFG.[2] The self dependency arcs lead to the constraints:

$$\begin{aligned}&\forall v \text{ in the RSFG, and } \forall k \in [0, k_v),\\ &A[k, v] - A[((k-1) \bmod k_v), v]\\ &\quad \geq T * \left\lfloor \frac{(k-1)}{k_v} \right\rfloor + r * d_v \end{aligned} \tag{4}$$

Now to obtain rate-optimal schedules, first we need to find the optimal computation rate. That is, to maximize $\frac{r}{T}$ subject to the above constraints (Inequalities (3) and (4)). An efficient solution for this problem can be obtained by relating it to the Minimum Cost-to-Time Ratio Cycle (MCTRC) problem in network flow [17, 18]. In [9] it was shown that the optimal computation rate can be computed by generating what is called the *precedence graph* from Inequalities (3) and (4). The precedence graph of an RSFG is much simpler than the corresponding homogeneous graph. Thus the optimal computation rate is derived from the precedence graph using any of the efficient algorithms for MCTRC [17]. Once the optimal $\frac{r}{T}$ is found, the value of various $A[k, v]$'s can be obtained by solving the linear equation with the optimal value of $\frac{r}{T}$. This can be reduced to the shortest path problem for which polynomial time solutions exist [17].

In the following subsection we extend the above formulation to MBRO schedules.

3.2. MBRO Schedules

We first show that the buffer requirement on each arc (u, v) in a RSFG can be represented as a linear constraint. Consider a time instant t such that node u is between its i-th and $(i+1)$-th firing for some i, and node v is between its i'-th and $(i'+1)$-th firing for some i'. Formally,

$$\begin{aligned}&\sigma(i, u) + d_u \leq t < \sigma(i+1, u) + d_u \quad \text{and}\\ &\sigma(i', v) + d_v \leq t < \sigma(i'+1, v) + d_v, \quad \text{for some } i, i'\end{aligned} \tag{5}$$

Since the i-th instance (i.e., the $(i+1)$-th firing) of u has finished, actor u would have produced $(i+1) * O_{uv}$ tokens on the arc (u, v). Further, as the i'-th instance of v has finished its execution, it must have consumed $(i'+1) * I_{uv}$ tokens from the arc (u, v). If there were m_{uv} tokens initially on the arc (u, v), then at time t, there would be $(i+1) * O_{uv} + m_{uv} - (i'+1) * I_{uv}$ tokens. Thus, the buffer size for the arc (u, v) at time t should be

$$b_{uv}(t) \geq (i+1) * O_{uv} + m_{uv} - (i'+1) * I_{uv} \tag{6}$$

We can represent the i-th instance of actor u as $(\lfloor \frac{i}{k_u} \rfloor, i \bmod k_u)$ in our notation. Representing $(i+1)$, i', and $(i'+1)$ in similar manner, and using them in Inequality (5), we obtain

$$\begin{aligned}&\left\lfloor \frac{\lfloor \frac{i}{k_u} \rfloor * T + A[(i \bmod k_u), u]}{r} \right\rfloor + d_u\\ &\quad \leq t < \left\lfloor \frac{\lfloor \frac{i+1}{k_u} \rfloor * T + A[(i+1) \quad \bmod k_u, u]}{r} \right\rfloor + d_u\end{aligned} \tag{7}$$

$$\begin{aligned}&\left\lfloor \frac{\lfloor \frac{i'}{k_v} \rfloor * T + A[(i' \bmod k_v), v]}{r} \right\rfloor + d_v\\ &\quad \leq t < \left\lfloor \frac{\lfloor \frac{i'+1}{k_v} \rfloor * T + A[(i'+1) \quad \bmod k_v, v]}{r} \right\rfloor + d_v\end{aligned} \tag{8}$$

After mathematical manipulations, we get

$$\begin{aligned}i &< \frac{(t+1-d_u) * r - A[k, u]}{T} * k_u + k \quad \text{and}\\ i'+1 &> \frac{(t-d_v) * r - A[k', v]}{T} * k_v + k'\end{aligned} \tag{9}$$

where $k = i \bmod k_u$, and $k' = (i'+1) \bmod k_v$. Since our estimate for the buffer size needs to be conservative, we substitute the upper bound for i and the lower bound for $i'+1$ (Inequality (9)) in Inequality (6).

$$\begin{aligned}&b_{uv}(t)\\ &\quad \geq \left(\frac{(t+1-d_u) * r - A[k, u]}{T} * k_u + k + 1\right) * O_{uv}\\ &\qquad - \left(\frac{(t-d_v) * r - A[k', v]}{T} * k_v + k'\right) * I_{uv} + m_{uv}\end{aligned} \tag{10}$$

By the property of multi-rate graphs, $k_u * O_{uv} = k_v * I_{uv}$. Using this in Inequality (10) and after simple algebraic manipulations, we get

$$\begin{aligned}&T * b_{uv}(t) + k_u * O_{uv} * (A[k, u] - A[k', v])\\ &\quad \geq (k+1) * T * O_{uv} - k' * T * I_{uv}\\ &\qquad + r * k_u * O_{uv} * (d_v - d_u + 1) + m_{uv} * T\end{aligned}$$

The right hand side of the above equation depends on the instance numbers k and k', and on constants k_u, k_v, O_{uv}, and I_{uv}. We will represent this as a function $\mathcal{C}(k, k')$. Note that the equation is linear in $b_{uv}(t)$, $A[k, u]$, and $A[k', v]$. Further the above equation is independent of t and hence

$$T * b_{uv} + (k_u * O_{uv}) * (A[k, u] - A[k', v]) \geq \mathcal{C}(k, k') \tag{11}$$

Lastly, the buffer estimate is independent of the iteration numbers of the actors u and v, but depends on k and k' which represent some instance of actors u and v respectively. More specifically, k is the most recent instance of actor u that has finished its firing. Likewise k' is the next instance of actor v that is to finish its firing. The difference in the interpretation of k and k' is due to the fact that $k = i \bmod k_u$ while $k' = (i' + 1) \bmod k_v$. For the buffer estimate to be valid, the above constraint must be satisfied for all values of $k \in [0, k_u)$ and for all values of $k' \in [0, k_v)$.

The MBRO problem can be formulated as follows. Our objective is to minimize the total buffer requirements for all arcs (i.e., $\sum b_{uv}$). This is the objective function in the MBRO formulation in Fig. 4. The schedule is specified by means of the $A[k, v]$ values and the computation rate r/T. The computed value of r/T using the Minimum Cost-to-Time Ratio Cycle (MCTRC) method [17] is used in all inequalities in the MBRO formulation to guarantee rate-optimality. The buffer requirement for each RSFG arc must satisfy Constraint I. For the schedule to satisfy dependency constraints, the value of $A[k, v]$'s must satisfy Constraint II in Fig. 4. Lastly, Constraint III ensures self dependency.

In the following subsection we will present a solution method for the MBRO formulation.

4. Solution to the MBRO Formulation and Discussions

The linear programming formulation can be solved using standard linear programming methods, such as the simplex method. Once the value of various $A[k, v]$'s are obtained, the buffer size for each arc needs to be calculated. Recall that in obtaining the MBRO formulation, we have used a conservative approach. Hence if we use Inequality (11) to compute the buffer requirements, the estimates could be more than what is required by the schedule. However, once the schedule is finalized buffer requirements on each arc can be computed using an operational method. This method essentially simulates the schedule to compute the buffer requirement. In particular, the algorithm considers the prologue and the repetitive pattern of the schedule and fires the actors accordingly. The algorithm also keeps track of the maximum number of tokens present on every arc during the execution of the repetitive pattern. These numbers give the buffer requirements for the individual arcs of the RSFG. From these values the total buffer requirements for all the arcs in the RSFG is computed.

4.1. Discussions

The MBRO formulation presented in the previous section is actually an integer program formulation as

[**MBRO Problem**] Minimize $\sum_{(u,v)} b_{uv}$ subject to the conditions

(I) $\forall(u, v), \forall k \in [0, k_u),$ and $\forall k' \in [0, k_v),$

$$T * b_{uv} + k_u * O_{uv} * (A[k, u] - A[k', v] \geq \mathcal{C}(k, k')$$

(II) $\forall(u, v),$ and $\forall k \in [0, k_v),$

$$A[k, v] - A\left[\left(\left\lceil \frac{(k+1) * I_{uv} - m_{uv} - O_{uv}}{O_{uv}} \right\rceil \bmod k_u\right), u\right] \geq$$

$$T * \left\lfloor \frac{1}{k_u} \left\lceil \frac{(k+1) * I_{uv} - m_{uv} - O_{uv}}{O_{uv}} \right\rceil \right\rfloor + r * d_u$$

(III) $\forall v,$ and $\forall k \in [0, k_v)$

$$A[k, v] - A[((k-1) \bmod k_v), v] \geq T * \left\lfloor \frac{(k-1)}{k_v} \right\rfloor + r * d_v$$

Figure 4. MBRO formulation.

the b_{uv} variables, which represent buffer sizes, must take integer values. The complexity of such an integer program formulation is NP-Complete. However since the buffer estimates in our formulation are conservative (refer to the discussion in the following paragraph), the solution obtained from the MBRO formulation (with or without the integer constraint on b_{uv} variables) may not always be optimal. Hence to reduce the complexity, we solve the MBRO formulation as a linear programming problem, omitting the integer constraints on b_{uv} variables. The obtained solution, although may not be optimal in terms of buffer requirements, is still a valid rate-optimal schedule as the data dependence constraints are satisfied at the optimal computation rate (r/T).

Next we discuss the two approximations that make the buffer estimates in our formulation conservative. First, we considered all combinations of (k, k') for Inequality (11). However, not all possible combinations of k and k' may arise in a schedule. For example, for arc (c, d) the following combinations do not arise at any time step in the repetitive pattern of the schedule shown in Table 3.

$$k = 2, \quad k' = 0 \quad \text{and} \quad k = 1, \quad k' = 1$$

Notice that our formulation estimates the buffer requirements even before the schedule is finalized and uses this estimate to arrive at a schedule. Since the schedule is not finalized at the time of formulation, it may not be possible, in general, to detect all combinations of (k, k') that may not arise. As a second approximation, we used larger values for i and smaller values for i' in Inequality (6) to estimate the buffer size.

Despite the conservative buffer estimates and the relaxation of integer constraints on b_{uv} variables, the MBRO formulation results in schedules that are buffer efficient compared to the MRSP formulation. That is, even though MBRO schedules are not buffer *optimal*, they use significantly less buffer storage compared to MRSP schedules. In Section 5.2 we compare the schedules obtained from the MBRO formulation and the MRSP formulation and show that except in a few cases (10 out of nearly 260 experiments), the MBRO formulation always produces buffer efficient schedules. Also, our experiments establish that solving the MBRO problem with the integer constraints leads to only a very small improvement (less than 0.5%) in buffer requirements.

4.2. Extension of MBRO Formulation

The proposed MBRO formulation does not take into account (processor) resource constraints. However it is possible to extend it to include resource constraints. The resulting formulation is a mixed integer linear program formulation. In [19, 20], we have formulated software pipelining (for homogeneous dataflow graphs) as an integer linear program problem. Also, this formulation considered only 1-periodic schedules. An interesting feature of this formulation is that it includes both processor resource constraints and buffer (register) constraints for rate-optimal scheduling. More recently, Fimmel and Muller have extended the software pipelining formulation for r-periodic schedules [21]. Further, these formulations [19–21] use the same scheduling form as in our MBRO formulation. In view of these, we feel that the extension of the resource constrained software pipelining formulation to multi-rate RSFGs should be easy. It should however be noted that the resulting formulation with resource constraints is an integer programming problem and is much harder to solve. The details of the formulation and its performance are beyond the scope of this paper.

5. Experimental Results

In this section we describe the implementation of our testbed and evaluate the proposed scheduling method. A comparison of the MBRO formulation with other existing methods is presented in Section 5.2.

5.1. The Scheduling Testbed

The experimental testbed is implemented using C programming language on a Unix platform. Further details on the implementation of the testbed can be found in [22]. The testbed consists of four main modules, namely the random graph generator, the module to generate and solve (for optimal r/T) the linear programming formulation for rate-optimal schedules, the module to generate and solve the linear programming problem with buffer minimization constraints, and the module to compute the buffer requirements for each arc in the RSFG.

Due to the lack of availability of standard multi-rate benchmark programs, and to extract general trends over a broad range of test inputs, we propose to evaluate our scheduling method on randomly generated well-behaved RSFGs. A similar approach was followed in other studies, e.g., in [14, 23]. The random graph generator ensures that the generated RSFG is *well-behaved* [4]. The following attributes of the generated RSFG are exponentially distributed random values: (i) the number of arcs per node, (ii) the number of tokens consumed/produced by a node on an input/output arc, and (iii) the execution time of each node. The mean value for the above parameters can be set independently, and are set to the average values observed in a certain DSP programs. Thus the random graph generator generates RSFGs that somewhat match in their characteristics to real DSP programs.

We supplement our performance study with results obtained from a set of DSP algorithms. The DSP algorithms considered in our experiments are; (1) phase-locked loop, (2) voice band modulation, (3) power spectrum, (4) auto-correlation, (5) periodogram, (6) comb filters, (7) satellite receiver, and (8) spectrum analyzer. The number of nodes and the number of arcs in each of these RSFGs are tabulated in Table 5.

The subsequent module in our testbed generates the linear constraints (corresponding to Inequalities (3) and (4)) of the MRSP formulation. The optimal computation rate $\frac{r}{T}$ is computed efficiently by relating the problem to the Minimum Cost-to-Time Ratio Cycle (MCTRC) problem. Using this $\frac{r}{T}$ value, the constraints for the MBRO problem are generated. The linear programming problem is solved using a public domain software package called *lp_solve*, developed in the Eindhoven University of Technology, The Netherlands.

In the testbed, we have also implemented an algorithm to compute the maximal buffer requirements for the arcs of the RSFG. The algorithm is based on an operational model in which the actors are fired according to the generated schedule. In particular, the algorithm considers the prologue and the repetitive pattern of the schedule and fires the actors accordingly. The algorithm also keeps track of the maximum number of tokens present on every arc during the execution of the repetitive pattern. These numbers give the buffer requirements on individual arcs of the RSFG. From these values the total buffer requirements for the RSFG is computed.

Table 5. Details of RSFGs for DSP applications.

Application	Number of nodes	Number of arcs
Phase locked loop	10	12
Voice band	7	14
Power spectrum	9	10
Auto correlation	16	17
Periodogram	7	7
Comb filter	21	26
Satellite receiver	19	25
Spectrum analyzer	6	7

5.2. *Experimental Results and Comparisons*

We compare the schedules generated by the MBRO formulation with those generated using (i) the MRSP formulation [9, 10], (ii) the block scheduling method [1], and (iii) the linear programming formulation based on homogeneous graphs [13].

5.2.1. Comparison with MRSP Schedules. In the first group of experiments, we evaluate how well the MBRO formulation minimizes the buffer requirements in comparison with the MRSP schedules [9]. The performance of the MBRO schedules is compared with the MRSP schedules for more than 260 randomly generated RSFGs, and the results for 10 candidate cases are shown in Table 6. For each random RSFG, we report the number of nodes and the number of arcs in the RSFG, the optimal computation period $\frac{T}{r}$, the buffer requirements in the MRSP and in the MBRO schedules, execution times (in CPU seconds) to solve the respective formulations on a Sun-SPARC workstation. We also report the percentage improvement achieved by the MBRO schedules with respect to the MRSP schedules computed as:

$$\%\text{ improvement} = \frac{\text{Buffer for MRSP} - \text{Buffer for MBRO}}{\text{Buffer for MRSP}} \quad (12)$$

Table 7 summarizes the comparisons for the considered DSP applications. Out of the 260 random RSFGs considered, in all but 10 cases, the MBRO schedules perform better than the MRSP schedules in terms of buffer requirements. In these experiments, we observe an average improvement of 24% in buffer requirements for the MBRO schedules. In certain cases, e.g., for the random RSFG with 28 nodes and 27 arcs, the improvement is as high as 51% and for the auto-correlation program it is 37%. The execution time of the MBRO schedule is comparable to that of the MRSP schedules for smaller RSFGs. But for larger RSFGs even though the execution time for the MBRO

Table 6. MRSP vs. MBRO schedules on random RSFGs.

		MRSP schedules			MBRO schedules			
No. of nodes	No. of arcs	Period	Buffer required	Execution time (CPU sec.)	Period	Buffer required	Execution time (CPU sec.)	% Buffer improvement
6	6	15	36	0.083	15	35	0.200	2.78
16	15	25	265	0.217	25	219	0.683	17.36
20	19	240	295	0.550	240	179	3.833	39.32
25	24	243	653	1.033	243	350	8.566	46.40
28	27	392	493	0.550	392	240	3.817	51.31
28	47	945	1646	2.100	945	1334	70.997	18.95
35	45	875	2278	2.433	875	1206	91.280	47.05
39	43	150	876	0.933	150	835	11.066	4.68
43	53	315	1586	2.050	315	1223	103.179	22.88
55	93	252	3895	3.400	252	3016	112.762	22.56

Table 7. MRSP vs. MBRO schedules on DSP applications.

	MRSP schedules			MBRO schedules			
Application program	Period	Buffer required	Execution time (CPU sec.)	Period	Buffer required	Execution time (CPU sec.)	% Buffer improvement
Phase locked loop	5	23	0.033	5	19	0.133	21.05
Voice band	16	23	0.083	16	23	0.133	0.00
Power spectrum	128	180	0.350	128	153	4.000	15.00
Auto correlation	256	364	2.700	256	229	120.812	37.08
Periodogram	256	245	2.100	256	197	104.462	19.59
Comb filter	2	33	0.117	2	33	0.133	0.00
Satellite receiver	151	625	3.333	151	427	295.172	31.68
Spectrum analyzer	84	53	0.083	84	47	0.217	11.32

formulation is significantly larger, it is still in the acceptable range of compilation time for application specific programs.

5.2.2. Comparison with Block Schedules. In the second group of experiments, we compare our scheduling method with the well-known block scheduling method of Lee and Messerschmitt [1]. The block scheduling method is based on an operational approach which uses the *earliest firing rule* [24]. The block schedules are also known as Periodic Admissible Parallel Schedules (PAPS). In all experiments, we assume the availability of infinite processor resources for the block schedules since the MBRO schedules are also based on the same assumption. The block scheduling method is applied to an RSFG unrolled a suitable number of times. In our experiments we unroll the RSFG by r times, where $\frac{r}{T}$ is the optimal computation rate in the irreducible form. A similar unrolling effect takes place when generating MBRO and MRSP schedules, even though the unrolling is never performed explicitly in either of the scheduling methods.

The results of both MBRO and the block scheduling methods for 10 randomly generated RSFGs are presented in Table 8. In this table, we report the computation period, buffer requirements, and execution time of the two scheduling methods. The columns on percentage improvement shows the improvement in computation period and buffer requirements calculated using an equation similar to Eq. (12). From Table 8, we observe that the MBRO schedules perform better than the block schedules in terms of both computation period

Table 8. Block schedules vs. MBRO schedules on random RSFGs.

		Block schedules			MBRO schedules				
No. of nodes	No. of arcs	Period	Buffer required	Execution time (CPU sec.)	Period	Buffer required	Execution time (CPU sec.)	% Buffer improvement	% Improvement in period
6	6	17	36	0.017	15	35	0.200	2.78	11.76
10	13	39	385	0.000	27	347	1.350	9.86	30.76
20	19	299	295	0.017	240	179	3.833	39.32	19.17
28	27	424	493	0.017	392	240	3.817	51.31	7.54
34	38	185	1135	0.033	139	797	13.716	29.78	24.86
37	46	385	1062	0.050	384	942	55.748	11.30	0.25
39	43	229	876	0.017	150	835	11.066	4.68	34.49
41	67	511	1686	0.050	240	1562	54.464	7.35	53.03
55	93	327	3637	0.050	252	3016	112.762	17.07	22.93
56	82	442	1693	0.067	243	1863	51.031	−8.36	45.02

Table 9. Block schedules vs. MBRO schedules on DSP applications.

	Block schedules			MBRO schedules				
Application program	Period	Buffer required	Execution time (CPU sec.)	Period	Buffer required	Execution time (CPU sec.)	% Buffer improvement	% Improvement in period
Phase locked loop	15	5	0.001	5	19	0.133	−42.42	66.67
Voice band	21	10	0.001	16	23	0.133	−36.11	23.80
Power spectrum	168	180	0.017	128	153	4.000	15.00	23.80
Auto correlation	559	156	0.083	256	229	120.812	−24.17	54.20
Periodogram	532	113	0.050	256	197	104.462	−29.89	51.87
Comb filter	12	7	0.001	2	33	0.133	−44.06	83.33
Satellite receiver	175	673	0.067	151	427	296.321	36.55	13.71
Spectrum analyzer	96	44	0.017	84	47	0.217	−6.82	12.50

and buffer requirements. In a block schedule, as the name indicates, the execution of nodes from different iterations are not overlapped. In fact, the execution of all nodes in one iteration must be completed before the execution of any node in the next iteration. This contributes to the increase in the period of block schedules. However, it is interesting to note that the lower computation rate of the block schedules do not bring any buffer storage improvement. The average improvement is 20% in computation period and 22% in buffer requirements.

In Table 9 we compare the block scheduling method with the MBRO method for the considered DSP application programs. For the DSP applications (refer to Table 9), the MBRO schedules perform better than the block schedules only in terms of computation rate. The buffer requirements for the block schedule are less than that for the corresponding MBRO schedules. However the computation period of the block schedule is significantly larger. In an MBRO schedule, since the operations from different iterations overlap, there is a good possibility that the MBRO schedules require more memory than the non-overlapped block schedules. Although the block scheduling method is simpler and computationally less expensive, it produces relatively poor schedules, in terms of the period of the schedules, primarily due to the non-overlapping nature of block schedules. Lastly, we observe that the compile time to construct the block schedules are significantly lower than that for MBRO schedules.

5.2.3. Comparison with Homogeneous Scheduling Methods. A linear programming formulation (called the Optimal Schedule Buffer Allocation (OSBA) formulation) to obtain rate-optimal buffer-optimal schedule for homogeneous graphs has been proposed in [13].

Unlike the MBRO formulation, the one proposed in [13] uses tight bounds for buffer estimates and therefore always generates schedules that have optimal buffer requirements. Hence it may be interesting to evaluate MBRO schedules against the schedules generated by the OSBA formulation. The OSBA formulation works on the equivalent homogeneous graph (e.g., the graph shown in Fig. 3), rather on the original RSFG itself (refer to Fig. 2). Once the optimal schedule is obtained from the OSBA formulation, buffer requirements for the arcs of the RSFG can be determined by using the operational method on the original RSFG. The reason for working with the RSFG, rather than with the homogeneous graph, in determining the buffer requirement, is as follows. In the RSFG, buffer space allocated for an arc is shared by all instances of the producing and consuming nodes. However, in the homogeneous graph, independent buffers need to be allocated for each homogeneous arc. Hence, combining the buffers for various homogeneous arcs that correspond to a single RSFG arc may lead to a reduction in buffer size as not all homogeneous arcs may have their 'live ranges' active at the same time. Combining is a form of buffer sharing among the homogeneous arcs that correspond to a single RSFG arc. In Section 6, we discuss a more general buffer sharing which facilities buffers to be shared among the arcs of the RSFG.

Tables 10 and 11 summarize the comparison between MBRO schedules and OSBA schedules. The computation rates of an OSBA schedule is equal to that of an MBRO schedule. However, surprisingly, in 75% of the test graphs used in our experiments, the MBRO schedules require lesser storage than the buffer-optimal schedules generated by the OSBA

Table 10. OSBA schedules vs. MBRO schedules on random RSFGs.

		OSBA schedules			MBRO schedules			
No. of nodes	No. of arcs	Period	Buffer required	Execution time (CPU sec.)	Period	Buffer required	Execution time (CPU sec.)	% Buffer improvement
10	13	27	373	2.483	27	347	1.350	6.96
16	15	25	223	1.367	25	219	0.683	1.79
20	19	240	187	5.316	240	179	3.833	4.27
25	24	243	423	9.783	243	350	8.566	17.25
28	27	392	279	4.983	392	240	3.817	13.97
35	45	875	1113	47.698	875	1206	91.280	−7.15
37	46	384	1029	44.048	384	942	55.748	8.45
39	43	150	775	14.899	150	835	11.066	−6.70
41	67	240	1646	51.531	240	1562	54.464	5.10
56	82	243	1884	46.215	243	1863	51.031	1.11

Table 11. OSBA schedules vs. MBRO schedules on DSP applications.

	OSBA schedules			MBRO schedules			
Application program	Period	Buffer required	Execution time (CPU sec.)	Period	Buffer required	Execution time (CPU sec.)	% Buffer improvement
Phase locked loop	5	19	0.133	5	19	0.133	0.00
Voice band	16	23	0.150	16	23	0.133	0.00
Power spectrum	128	173	2.000	128	153	4.000	11.56
Auto correlation	256	247	12.416	256	229	120.812	7.28
Periodogram	256	197	8.166	256	197	104.462	0.00
Comb filter	2	33	0.117	2	33	0.133	0.00
Satellite receiver	151	434	34.515	151	427	296.321	1.61
Spectrum analyzer	84	44	0.183	84	47	0.217	−6.82

formulation. This could be due to the following reason. The OSBA schedules are in fact buffer-optimal for the homogeneous graph; however, they may not be optimal when combining buffers that correspond to a single RSFG arc is allowed. Without such combining of buffers, the total buffer requirement for RSFGs in OSBA schedules could increase by 50% to 100%. This is the reason why we used the RSFGs in the first place to compute the buffer requirements. In our MBRO approach, buffer optimality constraints are specified taking into account this type of combining and their effects. As a consequence the objective function of the MBRO formulation is more effective in minimizing the buffer requirements. MBRO schedules have fewer memory requirement, by 9% on the average, compared to OSBA schedules.

5.2.4. Summary. We summarize the observations made from our experiments as follows.

- For the examples we used, the MBRO schedules perform significantly better than the MRSP schedules, in terms of buffer requirements. The improvement is 24% on the average and upto 51% in certain cases.
- Compared to Lee's block scheduling method, MBRO schedules perform 20% better in terms of the computation period for both random RSFGs and for the considered DSP applications. Despite a shorter computation period, MBRO schedules, in many test cases involving random RSFGs, showed a buffer improvement of 22%.
- The MBRO formulation obtains rate-optimal buffer minimized schedules directly from the RSFG. In 75% of our experiments the MBRO schedules require lesser storage (by 9%) than buffer-optimal OSBA schedules [13] obtained from the equivalent homogeneous graphs.
- Lee's block scheduling method requires the least amount of compile time (time to construct a schedule). The overall compile time for the MBRO and OSBA schedules do not differ significantly for the test cases we used, while that for the MRSP schedules are comparable to the block schedules.
- In a total of 260 experiments, only in 10 cases MBRO formulation produced schedules which are inferior, in terms of buffer requirements, to either MRSP schedules or block schedules. However, in all cases, the computation period of the MBRO schedules is significantly better than that of the block schedules.

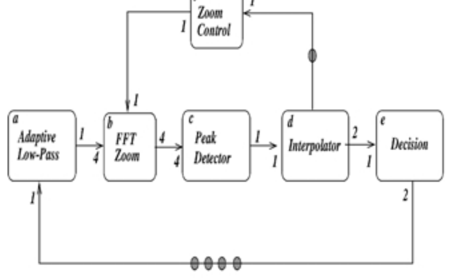
6. Reducing Memory Requirement by Buffer Sharing

In this section we present an approach to further reduce the memory requirement of the multi-rate dataflow graphs.

6.1. Buffer Sharing in RSFGs

Our approach here is to determine the live ranges of the various arcs, and share the allocated buffer space among the arcs whose live ranges do not overlap. Compared to the traditional graph coloring method for register allocation [25] where sharing is considered at a per register basis, in RSFGs sharing is to be considered at a per buffer basis, where each buffer may have a size greater than unity. We refer to this type of sharing as *complete-* or *full-buffer* sharing.

We illustrate our approach with the help of another example, the Spectrum Analyser algorithm considered in our experiment. The RSFG of the spectrum analyzer[3] is shown in Fig. 5(a). Let the execution time of actor



(a) RSFG for Spectrum Analyzer

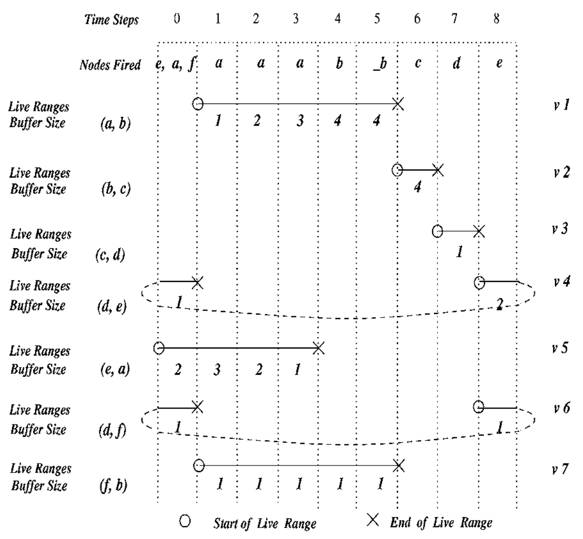
itera-tion	Time Step										
	0	1	2	3	4	5	6	7	8	9	10
0	a,f	a	a	a	b	_b	c	d	e	e	
1										a,f	a

(b) A Rate-Optimal Schedule

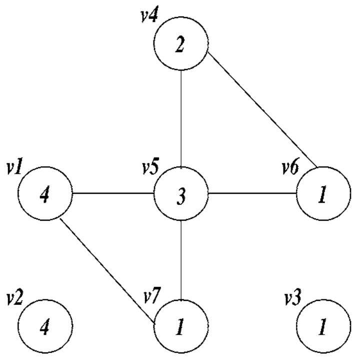
Buffer Requirement on Arcs							Total
(a,b)	(b,c)	(c,d)	(d,e)	(e,a)	(d,f)	(f,b)	
4	4	1	2	3	1	1	16

(c) Buffer Requirements for the Schedule

Figure 5. RSFG of spectrum analyser and its schedule.



(a) Live Ranges of Arcs



(b) Circular Arc Interval Graph

Figure 6. Live ranges of example arcs and its interference graph.

b be 2 time units while that of other actors be unity. A schedule for the RSFG is shown in Fig. 5(b) where the repetitive pattern appears from time step 2 to time step 9. The buffer requirements for the schedule are shown in Fig. 5(c).

Next, we define the live range of an arc. The live range of an arc starts at a point of time when the number of tokens in the associated buffer becomes greater than zero. The live range ends when the number of tokens in the associated buffer becomes zero again. The live range of each arc of the RSFG and the sizes of the associated buffers during the various time steps in the repetitive pattern are shown in Fig. 6(a). If the buffer associated with an arc contains at least one token throughout the repetitive pattern, we say that the live range of the arc spans the repetitive pattern. If the live range of an arc goes across iterations, we represent the live range by means of a circular arc [26]. As an example consider the live range of arc (d, e) which extends from time step 8 to 1. Finally, in an RSFG, it is possible that an arc may have multiple disjoint live ranges. In this paper, we associate a buffer to each live range rather than to each arc.

From Fig. 6(a), one can observe that the live ranges of arc (a, b) do not overlap with that of (b, c). Hence, instead of allocating individual buffer space for these two arcs, they can be allowed to share the same memory space. Further, the same buffer can also be shared by the live ranges of arc (c, d) and (d, e). Under buffer sharing, the total buffer size allocated for these live ranges is the maximum of the buffer size required for each of them, in this case 4. One can identify, for the given schedule (shown in Fig. 5(b)), the following buffer sharing is possible: Arcs (a, b), (b, c), (c, d), and (d, e) can share the same buffer space and requires a size of 4; arcs (d, f) and (f, b) can share the same buffer space and requires a size of 1; lastly arc (e, a) requires a buffer size of 3. Thus, the total memory requirement for the given schedule, under buffer sharing, is only 8. Thus buffer sharing significantly reduces the buffer requirement from 16 to 8.

In the following section we present a heuristic algorithm to perform buffer sharing for the arcs of an RSFG.

6.2. *A Heuristic Algorithm for Buffer Sharing*

As a first step, we determine the live ranges of the arcs of the RSFG for the given schedule. From this, we derive an interference graph $G = (V, E, w)$ where V and E represent respectively the vertices and edges of the interference graph. A vertex v represents a live range and an edge e between two vertices v and v' indicates that the respective live ranges overlap. The function w associates a weight $w(v)$, equals to the size of the buffer required for the corresponding live range, with vertex v. The interference graph for the example live ranges is shown in Fig. 6(b).

Now, the buffer sharing problem can be recognized as partitioning the set of vertices V into disjoint subsets $V_1, V_2, \ldots, V_m$ such that the vertices in each subset can share their buffers. The buffer requirement for a subset is determined by the live range in that subset that has the maximum buffer requirement. The objective of the partitioning is to minimize the sum of the buffer requirements of the subsets.

Our algorithm proceeds by first sorting the vertices of the interference graph in the descending order based on their weights. Starting with the first vertex (with maximum weight) in the sorted list we check every other vertex, whether the latter can share its buffer with the former. If the two vertices do not have an edge connecting them, then a subset including these two vertices is formed. We proceed by checking the remaining vertices, in the sorted order, if any other vertex can share its buffer with the current subset of vertices. For this, there should not be an edge between the new vertex and any of the vertices in the subset. If so, we include this vertex and proceed to check the remaining vertices. When all the vertices are examined, the current subset contains a (greedy) subset of vertices which can share their buffers. In a similar way, we can partition the remaining vertices iteratively. This approach is described formally in Algorithm 6.1.

Algorithm 6.1

Input: An interference graph $G = (V, E, w)$.
Output: The set of subsets $V_1, V_2, \ldots, V_m$ such that vertices in each V_k can share their buffers and with the objective to minimize the sum of the weights of the subsets.

```
[Step 1] Sort the vertices of the graph in the descending order of their weights.
[Step 2] Set $subset\_number = 0$;
[Step 3] For each vertex $v$ in the sorted list do
  [Step 3.1] If $v$ is already in some subset, go to Step 3.
  [Step 3.2] Increment $subset\_number$ by 1;
      Set $V_{subset\_number} = \{v\}$
  [Step 3.3] For each vertex $v'$ in the sorted list do
    [Step 3.3.1] If $v'$ is already in some subset, go to Step 3.3.
    [Step 3.3.2] For each $v''$ in $V_{subset\_number}$ do
      [Step 3.3.2.1] If there is an edge $(v', v'')$ in $E$, go to Step 3.3;
      /* The live range $v'$ overlaps with $v''$ and
         therefore $v'$ cannot share its buffer with those
         in $V_{subset\_number}$. */
    [Step 3.3.3] Include $v'$ in $V_{subset\_number}$. Go to Step 3.3.
  [Step 3.4] Set the weight of the subset $V_{subset\_number}$ to the weight of $w(v)$. Go to Step 3.
[Step 4] End.
```

We observe that our algorithm is a greedy algorithm and tries to share buffers for vertices with maximum weight first. When two or more vertices have the same weight, our algorithm considers these vertices in some order, typically the order in which they appear in the interference graph. More specifically, consider two vertices v_1 and v_2 having the same weight, i.e., $w(v_1) = w(v_2)$. Assume there exists an edge (v_1, v_2) in E. Let v be a vertex with a weight $w(v) > w(v_1)$. Further v does not share an edge with either v_1 or v_2. In deciding whether v_1 or v_2 is included in the same subset as v, our algorithm can make a bad choice, based on the order in which the nodes appear in the interference graph, which would preclude a number of other vertices being included in the same subset, thereby disallowing better buffer sharing. Thus, our algorithm may not lead to an optimal partitioning.

It should be noted here that though our buffer sharing problem resembles the traditional graph coloring problem of general or interval/circular arc graphs [25–29], there are two main differences: (i) the nodes in our interference graphs have associated weights; and (ii) we are interested in minimizing the sum of the weights of the subsets, rather than minimizing the number of subsets.

6.3. *Experimental Results*

We have implemented the buffer sharing algorithm on our testbed and experimented it on both randomly generated RSFGs and on the example DSP algorithms. In the performance comparison, the buffer sharing algorithm has been applied to all four scheduling methods, namely MRSP, MBRO, OSBA and Block scheduling methods. In the 260 experiments that we ran on the random RSFGs, the buffer sharing algorithm reduced the memory requirement of MBRO schedules in 231 cases (89%) by 5.2% (of the original memory requirement) on the average. In 6 of the 8 DSP applications a reduction in buffer requirement by 18.6% was achieved by our buffer sharing algorithm on the MBRO schedules.

Table 12. Comparison of scheduling methods with buffer sharing for random RSFGs.

	Comparison with MRSP schedules			Comparison with block schedules			Comparison with OSBA schedules		
	MRSP better	MBRO better	Both same	Block letter	MBRO better	Both same	OSBA better	MBRO better	Both same
No. of cases	57	187	16	1	147	112	175	81	2
Percentage improvement	13.23	21.48	–	62.0	21.12	–	13.90	8.04	–

Table 13. Comparison of scheduling methods with buffer sharing for DSP applications.

	Buffer requirement after sharing				% Improvement of MBRO over		
Application	MBRO	MRSP	Block schedules	OSBA	MRSP	Block schedules	OSBA
Phase-locked loop	19	22	14	18	15.79	−26.32	−5.26
Voice-band	22	22	24	24	0.00	9.09	9.09
Power spectrum	153	164	132	164	7.19	−13.73	7.19
Auto correlation	227	292	172	171	28.63	−24.23	−24.67
Periodogram	197	245	130	197	24.37	−34.01	0.00
Comb filter	23	24	13	23	4.35	−43.48	0.00
Satellite receiver	386	523	578	332	35.50	49.74	−13.99
Spectrum analyzer	28	34	37	25	21.43	32.14	−10.71

The result of applying the buffer sharing algorithm for MRSP, Block, and OSBA scheduling methods are reported in Tables 12 and 13. Even with buffer sharing, MBRO schedules resulted in fewer memory requirement than MRSP schedules in a large majority of cases. More specifically, the improvement was in 187 test cases, with an average improvement in buffer requirement of 21.5%. However, the number of cases where MRSP schedules had lower memory requirement (by 13.2% on the average) compared to MBRO schedules has increased to 57 (out of 260) random RSFGs. This could be due to the fact that the buffer minimization constraint in our MBRO formulation, though minimizes the buffer requirement (before buffer sharing), may result in schedules where the overlap of the live ranges can be to a greater extent, reducing the extent of buffer sharing. It should however be noted that the MBRO method performs better overall, as in nearly 72% of the test cases, MBRO schedules had fewer buffer requirements. Further the percentage improvement in buffer requirement in MBRO schedules is also higher (21.48%) compared to MRSP schedules (13.23%). Lastly in all 8 DSP applications, MBRO schedules had fewer or equal buffer requirements compared to MRSP schedules (refer to Table 13).

Buffer sharing in OSBA leads to a more significant improvement than MBRO. More specifically, in more than 67% of the random RSFGs, OSBA schedules had fewer buffer requirement, with an average improvement of 13.9%. MBRO schedules performed better in 31% cases with an average improvement of 8%. It should be noted here that the OSBA algorithm is based on first obtaining the homogeneous graphs. As a consequence of this, the complexity of the OSBA algorithm is higher than that of the MBRO algorithm. Thus we observe that, with buffer sharing, OSBA algorithm performs better than the MBRO method in a large majority of cases. Further study is required to see if the specifics of the buffer sharing algorithm has any effect on the above result. The improvement in buffer requirement of block schedules over MBRO after buffer sharing in Table 13 should be considered keeping in mind that the computation of period of block schedules are, in general, much higher to that of MBRO schedules. Lastly, we remark that the performance comparison of the scheduling methods without buffer sharing reported in Tables 6 to 11 in Section 5 are interesting in their own right, as buffer sharing may (i) introduce additional complexity in code generation and (ii) require additional hardware support in the form of FIFO buffer pointers.

7. Related Work

Our earlier work on rate-optimal schedules [9] deals only with (computation) rate-optimality; it does not consider minimizing buffer requirement. This paper extends the above work to choose, among the various rate-optimal schedules, the schedule that has less buffer space requirement.

In [1], a static scheduling method for synchronous dataflow graphs has been presented by Lee and Messerschmitt. Lee's method, which is based on the operational model, constructs non-overlapped schedules and hence may not always lead to rate-optimality. Further this scheduling does not consider reducing buffer requirements. However, the block scheduling method can handle processor resource constraints which are not considered by either MRSP or MBRO methods. Further, several other scheduling methods, some of them based on the block scheduling method, with different objectives have been proposed by Lee's research group. For example, methods to construct compact loop schedules, called single-appearance schedules have been proposed in [30]. Sih and Lee have studied the scheduling of acyclic synchronous dataflow graphs taking into consideration the interprocessor communication delays, another issue not addressed by MRSP or MBRO formulation. They have proposed several solution methods for scheduling the acyclic graphs [14]. Compile-time scheduling of dataflow program graphs with dynamic constructs was reported in [31].

In [32], algorithms to construct either space-optimal or maximally-concurrent schedules have been discussed. Their method is based on the operational model and does not overlap operations belonging to successive iterations. Parhi and Messerschmitt [33] have studied the static rate-optimal scheduling of iterative dataflow flow programs via optimal unfolding. However, their method has focused *only* on homogeneous dataflow graphs and does not try to optimize buffer space requirements. Lucke and Parhi [34] have proposed a novel resource-constrained scheduling method for homogeneous graphs. Their approach is based on integer programming and can optimally unfold, retime, and pipeline the graph to achieve rate-optimal schedules with minimum processor requirement. The method presented by Chao and Sha [35] to obtain the optimal computation rate for a homogeneous dataflow graph is somewhat similar to Parhi's method [33], but is capable of obtaining either a pipelined or a non-pipelined schedule. Jeng and Chen [11] have proposed a novel unfolding technique to construct rate-optimal schedules with the smallest unfolding factor.

Leiserson et al., proposed a method based on 'retiming' for synchronous circuits [36]. Although the semantics of synchronous circuits is different from that of RSFGs, it is possible to use their approach by first converting the RSFGs into homogeneous dataflow graphs. However, as discussed in Section 5.2.3, a memory requirement minimization method (OSBA) [13] based on homogeneous graphs, yields inferior schedules, in terms of buffer requirement, in a large percentage (more than 90%) of the test cases. As the minimization methods based on homogeneous graphs do not capture the notion of combining buffers (of homogeneous arcs that correspond to a single RSFG arc), the buffer requirement of the resulting schedules is higher, though only by a small percentage (by 9% on the average).

Printz [2] discusses a method to map large-grain synchronous signal processing programs onto multiprocessor machines using a style which combines serial, systolic, and data-parallel implementations. The scheduling and mapping obtained using this method are not optimal, though it is guaranteed that the constructed schedule is within a certain factor from the optimality. A complete DSP design environment, called McDAS was developed by Hoang [37]. McDAS consists of a scheduler which is based on the *iterative clustering* approach. A method to construct compact looped schedules for a variant of synchronous dataflow graphs, called scalable synchronous dataflow graphs, has been proposed by Ritz et al. [38]. Related work applied in the areas of high-level synthesis and code generation can be found in [3, 12]. In [39], an integer linear programming formulation is used to derive rate-optimal resource constrained schedules in high-level synthesis applications. This method is targeted towards acyclic graphs and can support r-periodic schedules only by explicitly unfolding the graph. However, this formulation takes care of (processor) resource-constraints. A wealth of literature has been reported in the area of software pipelining for general-purpose computations [6, 7]. The reader is referred to [8] for a survey of these work.

A number of rate-optimal scheduling methods for software pipelining [6–8] have been proposed in the literature that also assume unlimited resources [13, 24, 33, 34, 40, 41]. But these work deal only with homogeneous dataflow graphs [5] and do not work directly for large grain synchronous data flow graphs or RSFG model [1, 4]. Our work [9, 10], for the first time, formulated the rate-optimal scheduling problem for multi-rate computation directly. In this paper, we have extended the formulation for minimizing buffer requirements as well. Lastly, the formulations presented in [19–21, 42, 43] deal with resource constrained rate-optimal software pipelining for homogeneous dataflow graphs.

Retiming [36] has been used as a method for obtaining software pipelined schedules in [44, 45]. The notion of two-dimensional retiming has been introduced in [46]. Subsequently two-dimensional retiming has been used to derive schedules that meet a given throughput while reducing the memory requirements [47]. Denk and Parhi also propose efficient solutions for memory requirement minimization in statically scheduled DSP programs [48]. They consider different memory models, viz., operation-constrained, processor-constrained, and unconstrained memory models, with and without simultaneous pipelining or retiming. It should be noted here that these approaches are targeted for homogeneous dataflow graphs, whereas the MBRO formulation proposed in this paper is for multi-rate RSFGs.

The buffer sharing algorithm considered in this paper closely resembles graph coloring or identifying maximal independent sets in interval graphs [27]. Efficient algorithms for finding maximal independent sets for subclasses of *perfect graphs* have been proposed [28, 29]. Chaitin et al. have used interval graph coloring method to solve the register allocation problem in compilers [25]. However, these work concentrate only on unweighted graphs and, as noted in Section 6.2, are somewhat different from our buffer sharing problem. In [49], a buffer merging technique has been proposed to reduce the buffer requirements of single appearance block schedules[4] [30]. Their approach is based on analyzing the buffer requirements of the arcs of the graph in a systematic way using efficient polynomial time heuristics. Their earlier work also considered deriving single appearance block schedules that has minimum buffer requirements, where buffer sharing is not considered [50]. Buffer minimization through an evolutionary approach for single appearance block schedules has been presented in [51]. Ritz et al. give an enumerative approach for reducing data buffer memory in synchronous data flow graphs [52]. Again their approach is for single appearance block schedules. In contrast the buffer minimization proposed in this paper is for software pipelined or overlapped schedules.

8. Conclusions

In this paper we have studied the problem of constructing rate-optimal schedules which also minimizes buffer requirements of the constructed schedules. The problem has been formulated, as a linear programming problem called the MBRO formulation. A unique feature of our method is that it directly constructs schedules which minimize buffer requirement for the optimal computation rate. The construction process also gives the optimal unfolding factor for rate-optimal schedules. Further, in the proposed schedules the execution of nodes from successive iterations do overlap, exploiting software-pipelining. Lastly we have proposed a greedy heuristic algorithm to perform buffer sharing among the arcs of the RSFG to further reduce the memory requirement of the MBRO schedules.

The proposed scheduling method is implemented and a scheduling testbed is constructed to evaluate the proposed method on a set of DSP applications and on a set of randomly generated RSFGs. In addition to the MBRO formulation, the rate-optimal multi-rate scheduling method (MRSP) [9], the Optimal Scheduling and Buffer Allocation (OSBA) method [13], and the Periodic Admissible Parallel Schedule method (PAPS) [1] have also been implemented. In a large majority of cases, the MBRO schedules perform significantly better than the MRSP and the PAPS schedules in terms of buffer requirements. Also, the MBRO schedules perform considerably better than the PAPS schedules in terms of computation rate. The total execution time for the MBRO method is comparable to MRSP or OSBA scheduling methods. Lastly, the proposed buffer sharing algorithm further reduces the memory requirement of MBRO schedules in a large majority of the test cases.

Acknowledgments

The authors are thankful to Prof. Edward A. Lee and Prof. Shuvra Bhattacharyya for their suggestions in the comparative study of various scheduling methods. Further some of the DSP applications reported in this paper are taken from Ptolemey: An Environment for Heterogeneous System Simulation and Prototyping, developed in the University of California, Berkeley. The authors thank Dr. V. Živojnović and Dr. S. Ritz for their contribution in modeling the spectrum analyzer and the satellite receiver applications. The authors acknowledge Natural Science and Engineering Research Council (NSERC) and MICRONET, Network of Centers of Excellence for supporting this work. The support received from the members of ACAPS (Advanced Computer Architecture and Program Structures Group) at McGill University is acknowledged. Lastly, the authors are thankful to the anonymous reviewers for their constructive comments.

Notes

1. Throughout this paper we do not consider processor resource constraints while constructing optimal rate schedules. A discussion on including processor resource constraints is presented in Section 4.2.
2. This assumption and the corresponding constraint (Inequality 4) can be removed if desired, and hence is not a limitation of our formulation. However, when relaxing this assumption one must ensure that T is greater than 0, as there may not be any recurrence cycle in the RSFG. Typically, in such cases, the values of T and r are set to 1 in software pipelining.
3. For brevity, we have scaled the number of tokens produced/consumed by a node in the spectrum analyzer example. However, tasks such as Peak Detector, will consume as many as 256 input samples to produce a single output.
4. In a single appearance schedule, each node appears exactly once with a repetition count. A set of nodes may be grouped together (nesting) and a repetition count can be associated with the grouping/nesting. Since each node (and hence the code for each node) appears only once in the schedule, the software synthesis of the schedule is compact and efficient in terms of program memory.

References

1. E. Ashford Lee and D.G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing, *IEEE Transactions on Computers*, vol. 36, no. 1, 1987, pp. 24–35.
2. H. Printz, "Automatic Mapping of Large Signal Processing Systems to a Parallel Machine," Ph.D. Thesis. Published as Memorandum CMU-CS-91-101, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1991.
3. S. Ritz, M. Pankert, and H. Meyr, "High Level Software Synthesis for Signal Processing Systems," in *Proceedings of the 1992 International Conference on Application Specific Array Processors*, Berkeley, California, Aug. 4–7, 1992.
4. G.R. Gao, R. Govindarajan, and P. Panangaden, "Construction Rules for Well-Behaved Stream Programs," ACAPS Technical Memo 26, School of Computer Science, McGill University, Montréal, Québec, April 1992.
5. J.B. Dennis, "First Version of a Data-Flow Procedure Language," in *Proceedings of the Colloque sur la Programmation*, Lecture Notes in Computer Science, vol. 19, Berlin: Springer-Verlag, 1975, pp. 362–376.
6. B.R. Rau and C.D. Glaeser, "Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing," in *Proceedings of the 14th Annual Microprogramming Workshop*, Chatham, Massachusetts, Oct. 12–15, 1981, pp. 183–198.
7. M. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," in *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, Atlanta, Georgia, June 22–24, 1988, pp. 318–328.
8. V.H. Allan, R.B. Jones, R.M. Lee, and S.J. Allan, "Software Pipelining," *ACM Computing Surveys*, vol. 27, no. 3, 1995, pp. 367–432.
9. R. Govindarajan and G.R. Gao, "A Novel Framework for Multi-Rate Scheduling in DSP Applications," in *Proceedings of the 1993 International Conference on Application Specific Array Processors*, Venice, Italy, Oct. 25–27, 1993, pp. 77–88.
10. R. Govindarajan and G.R. Gao, "Rate-Optimal Schedule for Multi-Rate DSP Computations," *Journal of VLSI Signal Processing*, vol. 9, no. 3, 1995, pp. 211–232.
11. L.-G. Jeng and L.-G. Chen, "Rate-Optimal DSP Synthesis by Pipeline and Minimum Unfolding," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 2, no. 1, 1994, pp. 81–88.
12. D.B. Powell, E.A. Lee, and W.C. Newman, "Direct Synthesis of Optimized DSP Assembly Code from Signal Flow Block Diagrams," in *Proceedings of ICASSP-92, the 1992 International Conference on Acoustics, Speech, and Signal Processing*, San Francisco, California, March 23–26, 1992, pp. 553–556.
13. Q. Ning and G.R. Gao, "A Novel Framework of Register Allocation for Software Pipelining," in *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Charleston, SC, Jan. 10–13, 1993, pp. 29–42.
14. G.C. Sih, "Multiprocessor Scheduling to Account for Interprocessor Communication," Ph.D. Thesis, Memorandum no. UCB/ERL M91/29, Electronics Research Laboratory, University of California at Berkeley, 1991.
15. E.A. Lee, "Consistency in Dataflow Graphs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, 1990, pp. 223–235.
16. R. Reiter, "Scheduling Parallel Computations," *J. of the ACM*, vol. 15, no. 4, 1968, pp. 590–599.
17. E.L. Lawler, *Combinatorial Optimization: Networks and Matroids*, Ft. Worth, Texas: Saunders College Publishing, 1976.
18. R.M. Karp, "A Characterization of the Minimum Cycle Mean in a Digraph," *Discrete Mathematics*, vol. 23, 1978, pp. 309–311.
19. R. Govindarajan, E.R. Altman, and G.R. Gao, "A Framework for Resource-Constrained Rate-Optimal Software Pipelining," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 11, 1996, pp. 1133–1149.
20. E.R. Altman, R. Govindarajan, and G.R. Gao, "A Unified Framework for Instruction Scheduling and Mapping for Function Units with Structural Hazards," *Journal of Parallel and Distributed Computing*, vol. 49, no. 2, 1998, pp. 259–294.
21. D. Fimmel and J. Muller, "Optimal Software Pipelining under Resource Constraints," Technical Report SFB 358-A1-1/99, Dresden University of Technology, Germany, 1999. Available at `http://www.iee.et.tu-dresden.de/~desa/A1/Pipeline.ps.zip`.
22. P. Desai, "An Implementation of a Multi-Rate Scheduling Testbed," ACAPS Technical Note, School of Computer Science, McGill University, Montreal, Québec, 1993.

23. G. Liao, E.R. Altman, V.K. Agarwal, and G.R. Gao, "A Comparative Study of DSP Multiprocessor List Scheduling Heuristics," in *Proceedings of the International Hawaii System Science Conference*, Hawaii, 1994.
24. G.R. Gao, Y.-B. Wong, and Q. Ning, "A Petri-Net Model for Fine-Grain Loop Scheduling," ACAPS Technical Memo 18, School of Computer Science, McGill University, Montréal, Québec, Jan. 1991.
25. G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P.W. Markstein, "Register Allocation via Coloring," *Computer Languages*, vol. 6, 1981, pp. 47–57.
26. L.J. Hendren, G.R. Gao, E.R. Altman, and C. Mukerji, "A Register Allocation Framework Based on Hierarchical Cyclic Interval Graphs," in *Proceedings of the 4th International Conference on Compiler Construction, CC '92*, Paderborn, Germany, Oct. 5–7, 1992, U. Kastens and P. Pfahler, (Ed.), Lecture Notes in Computer Science, vol. 641, Berlin: Springer-Verlag pp. 176–191.
27. M.C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, New York: Academic Press, 1980.
28. T. Kashiwabara, S. Masuda, K. Nakajsma, and T. Fujisawa, "Generation of Maximum Independent Sets of a Bipartite Graph and Maximum Cliques of a Circular Arc Graph. Interval, Circular-Arc and Chordal Graphs," *Journal of Algorithms*, vol. 13, 1992, pp. 161–1745.
29. J.Y.T. Leung, "Fast Algorithms for Generating All Maximum Independent Sets of Interval, Circular-Arc and Chordal Graphs," *Journal of Algorithms*, vol. 5, 1984, pp. 22–35.
30. S.S. Bhattacharyya and E.A. Lee, "Scheduling Synchronous Dataflow Graphs for Efficient Looping," *Journal of VLSI Signal Processing*, vol. 6, no. 3, 1993.
31. S. Ha and E.A. Lee, "Compile-Time Scheduling and Assignment of Data-Flow Program Graphs with Data-Dependent Iteration," *IEEE Transactions on Computers*, Vol. 40, no. 11, 1991, pp. 1225–1238.
32. M. Čubrić and P. Panangaden, "Minimal Memory Schedules for Dataflow Networks," In *Proceedings of the 4th International Conference on Concurrency Theory*, Hildesheim, Germany, Aug. 23–26, 1993, Eike Best (Ed.), Lecture Notes in Computer Science, vol. 715, Berlin: Springer-Verlag, pp. 368–383.
33. K.K. Parhi and D.G. Messerschmitt, "Static Rate-Optimal Scheduling of Iterative Data-Flow Programs via Optimum Unfolding," *IEEE Transactions on Computers*, vol. 40, no. 2, 1991, pp. 178–195.
34. L.E. Lucke and K.K. Parhi, "Generalized ILP Scheduling and Allocation for High-Level DSP Synthesis," in *Proceedings of the 1993 IEEE Custom Integrated Circuits Conference*, 1993.
35. L.-F. Chao and E.H.-M. Sha, "Rate-Optimal Static Scheduling for DSP Dataflow Programs," in *Proceedings of 1993 Great Lakes Symposium on VLSI*, March 1993, pp. 80–84.
36. C.E. Leiserson, F.M. Rose, and J.B. Saxe, "Optimizing Synchronous Circuitry by Retiming," in *Proceedings of the Third Caltech Conference on VLSI*, Pasadena, California, March 1983, pp. 87–116.
37. P.D. Hoang, "Compiling Real-Time Digital Signal Processing Applications onto Multiprocessor Systems," Ph.D. Thesis, Memorandum no. UCB/ERL M92/68, Electronics Research Laboratory, University of California at Berkeley, 1992.
38. S. Ritz, M. Pankert, V. Živojnović, and H. Meyr, "Optimum Vectorization of Scalable Synchronous Dataflow Graphs," in *Proceedings of the 1993 International Conference on Application Specific Array Processors*, Venice, Italy, Oct. 25–27, 1993, pp. 285–296.
39. C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu, "A Formal Approach to the Scheduling Problem in High-Level Synthesis," *IEEE Transactions on Computer-Aided Design*, vol. 10, no. 4, 1991, pp. 464–475.
40. V. van Dongen, G.R. Gao, and Q. Ning, "A Polynomial Time Method for Optimal Software Pipelining," in *Proceedings of the Conference on Vector and Parallel Processing, CONPAR-92*, Lyon, France, Berlin: Springer-Verlag, Sept. 1–4, 1992, Lecture Notes in Computer Science, vol. 634, pp. 613–624.
41. J. Ramanujam, "Optimal Software Pipelining of Nested Loops," in *Proceedings of the 8th International Parallel Processing Symposium*, Cancún, Mexico, April 26–29, 1994, pp. 335–342.
42. F. Depuydt, W. Geurts, G. Goossens, and H. De Man, "Optimal Scheduling and Software Pipelining of a Repetitive Signal Flow Graphs with Delay Line Optimization," in *Proc. of the EDDA European Design and Test Conference*, Paris, France, Feb. 1994.
43. A.E. Eichenberger and E.S. Davidson, "Efficient Formulation for Optimal Modulo Schedulers," in *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, Las Vegas, Nevada, June 15–18, 1997, pp. 194–205.
44. M. Potkonjak and J. Rabey, "Retiming for Scheduling," in *Proceedings of the VLSI Signal Processing Workshop*, 1990.
45. P.-Y. Calland, A. Darte, and Y. Robert, "Circuit Retiming Applied to Decomposed Software Pipelining," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 1, 1998, pp. 24–35.
46. N. Passos, E.H.-M. Sha, and S. Bass, "Scheduled-Based Multi-Dimensional Retiming on Data Flow Graphs," in *Proceedings of the 8th International Parallel Processing Symposium*, Cancún, Mexico, April 26–29, 1994, pp. 195–199.
47. T.C. Denk, M. Majumdar, and K.K. Parhi, "Two-Dimensional Retiming with Low Memory Requirements," in *Proceedings of ICASSP-96, the 1996 International Conference on Acoustics, Speech, and Signal Processing*, Atlanta, Georgia, May 1996, pp. 3330–3333.
48. T.C. Denk and K.K. Parhi, "Lower Bounds on Memory Requirements for Statically Scheduled DSP Programs," *Journal of VLSI Signal Processing*, vol. 12, 1996, pp. 247–264.
49. P.K. Murthy and S.S. Bhattacharyya, "A Buffer Merging Technique for Reducing Memory Requirements of Synchronous Dataflow Specifications," in *Proceedings of the International Symposium on Systems Synthesis*, San Jose, CA, Nov. 1999.
50. P.K. Murthy, S.S. Bhattacharyya, and E.A. Lee, "Joint Minimization of Code and Data for Synchronous Data Flow Programs," *Journal on Formal Methods in System Design*, vol. 11, no. 1, 1997, pp. 41–70.
51. E. Teich, J. Zitzler, and S.S. Bhattacharyya, "Buffer Memory Optimization in DSP Applications—An Evolutionary Approach," in *Proceedings of the International Conference on Parallel Problem Solving from Nature*, Amsterdam, The Netherlands, Sept. 1999, pp. 885–894.

52. S. Ritz, M. Willems, and H. Meyr, "Scheduling for Optimum Data Memory Compaction in Block Diagram Oriented Software Synthesis," in *Proceedings of ICASSP-95, the 1995 International Conference on Acoustics, Speech, and Signal Processing*, 1995.

R. Govindarajan received his B.Sc. degree from Madras University in 1981 and B.E. (Electronics and Communication) and Ph.D. (Computer Science) degrees from the Indian Institute of Science in 1984 and 1989. He held post-doctoral research positions at the University of Western Ontario, London, Ontario, Canada and McGill University, Montreal, Québec, Canada. He was a faculty at the Department of Computer Science, Memorial University of Newfoundland, St. John's, Newfoundland, Canada between 1994–95. Since then he has been with the Supercomputer Education and Research Centre and the Department of Computer Science and Automation, Indian Institute of Science, where he is now an Associate Professor. Govindarajan's current research interests are in the areas of Compilation techniques for Instruction-Level Parallelism, Digitial Signal Processing, Compilation Techniques for DSP and Embedded Processors, Distributed Shared Memory Architectures, and Cluster Computing. R. Govindarajan is a Senior Member of the IEEE, and a Member of IEEE Computer Society and ACM SIGARCH.
govind@serc.iisc.ernet.in

Guang R. Gao received his S.M. and Ph.D. degrees in Electrical Engineering and Computer Science from the Massachusetts Institute of Technology, in 1982 and 1986, respectively. Currently he is a professor at the Department of Electrical and Computer Engineering at the University of Delaware, where he has been the founder and leader of the Computer Architecture and Parallel Systems Laboratory. Prior to that he has been an Associate Professor of the School of Computer Science, McGill University, Montreal, Canada. Prof. Gao's research interests include high-performance computing systems and architectures, programming language design and implementation, parallel programming and applications. Prof. Gao has over 100 research publications in refereed conference/workshop proceedings and journals. He was the Co-Editor of Journal of Programming Languages, and a member of the Editorial Board of the IEEE Concurrency journal and IEEE Transaction on Computers. He has been a program/general chair, or a member of steering/program/organizing committee of many international conferences in his field (e.g., IEEE International Symposium of High-Performance Computer Architecture, ACM International Conference on Supercomputing, ACM/IEEE International Symposium on Microarchitectures, IFIP and ACM SIGARCH International Conference on Parallel Architectures and Compilation Techniques, Parallel Architecture and Language Europe, and EURO-PAR Conference, High Performance Computing Symposium (HPCS) and the International Compiler Construction Conference (CC). He has edited or co-edited several research monographs. He has served as a Guest Editor on Special Issues for the Journal of Parallel and Distributed Computing and IEEE Transaction on Computers. Currently, Dr. Gao is a Distinguished Visitor and a Senior Member of IEEE CS, and a member of ACM SIGARCH, SIGPLAN and IFIP WG 10.3.
ggao@eecis.udel.edu

Palash Desai is a product marketing for Conductus, Inc. Prior to joining Conductus, Palash was an associate with Pittglio, Rabin, Todd & McGrath (PRTM), and a design engineer with Lucent Technologies in the Wireless Networks Unit. Palash holds a Bachelor and Master of Engineering from McGill University, Montréal, Canada, and a Master of Business Administration from the Johnson Graduate School of Management, Cornell University, Ithaca, USA.
Palash.Desai@Conductus.com