McGill University
School of Computer Science
ACAPS Laboratory

Advanced Compilers, Architectures and Parallel Systems

Rate-Optimal Schedule for Multi-Rate DSP Computations

R. Govindarajan
Guang R. Gao

ACAPS Technical Memo 61
August 16, 1993

govind@cs.mcgill.ca, gao@cs.mcgill.ca

Abstract

In this paper, we present a novel framework of *multi-rate* scheduling of signal processing programs represented by regular stream flow graphs (RSFGs). The main contribution of this paper is translating the scheduling problem of RSFGs into an equivalent problem in the domain of Karp-Miller computation graphs. A distinct feature of our scheduling framework — called *multi-rate software pipelining* — is to allow maximum overlapping of operations from successive iterations subject only to precedence constraints caused by data dependences. Our framework combines the insights and experience from two seemingly different lines of research: static scheduling of application-specific programs and software pipelining of loops in general-purpose parallelizing compilers.

We demonstrate that the scheduling of regular stream flow graphs can be formulated as a mathematical problem by capturing data dependencies between two actors as a precedence relation between the firing of these actors. Using linear schedules, the problem is further translated into a linear programming formulation. An efficient solution for the linear programming problem is obtained by first constructing what is called the *precedence graph*. The precedence graphs are similar to the Karp-Miller computation graphs. A polynomial-time solution is obtained by observing that the computation rate of the optimal schedule is the *minimum cost-to-time ratio cycle* (MCTRC) in the precedence graph and using the well-established solution methods for the MCTRC problem. Finally, a graph coloring method based on the cyclic interval graph representation has been proposed to minimize the buffer space requirement for rate-optimal schedules.

Contents

1 Introduction **1**

2 Regular Stream Flow Graphs **4**
- 2.1 Operational Model of Regular Stream Flow Graphs 4
- 2.2 State Equations 5
- 2.3 Timed Stream Flow Graphs 7

3 Motivating Example **7**

4 Multi-Rate Software Pipelining (MRSP): Problem Formulation **12**
- 4.1 Admissible Schedule 13
- 4.2 Periodic Schedules 16
- 4.3 Precedence Constraints in the RSFG 17
- 4.4 Problem Formulation 19
- 4.5 Back to Motivating Examples 20

5 Solution to the MRSP Problem **22**
- 5.1 Construction of Equivalent Precedence Graph 22
- 5.2 An Efficient Algorithm for MCTRC 24

6 An Approach to Minimize Buffer Space **26**

7 Related Work **29**
- 7.1 Scheduling in DSP Architectures and Applications 29
- 7.2 Loop Scheduling in Parallelizing Compilers 30

8 Conclusions and Future Work **31**

9 Acknowledgments **32**

A Transformation of RSFGs into Homogeneous Dataflow Graphs **36**

B Block Scheduling of RSFGs **38**
- B.1 Non-Overlapped Schedule for Example 3.1 38
- B.2 Non-Overlapped Schedule for Example 3.2 38

List of Figures

- 1 An Example RSFG 6
- 2 A Rate-Optimal Schedule 8
- 3 Equivalent Homogeneous Graph for Example 3.1 9
- 4 Regular Stream Flow Graph of Example 3.2 10
- 5 A Rate-Optimal Schedule 11
- 6 Equivalent Homogeneous Graph for Example 3.1 with 6 Tokens on (b, a) 12
- 7 An Arc in the RSFG 13
- 8 Homogeneous Precedence Graphs 25
- 9 Live Ranges of Arcs of the Homogeneous Graph 28
- 10 A Non-Overlapped Schedule for the RSFG in Example 3.1 38
- 11 A Non-Overlapped Rate-Optimal Schedule for Example 3.1 39
- 12 A Non-Overlapped Schedule for the RSFG of Example 3.2 39
- 13 A Non-Overlapped Schedule for the RSFG after Retiming 40

1 Introduction

Digital Signal Processing (DSP) applications present both challenges and opportunities for architects of high-performance systems. As the amount of parallelism available in VLSI DSP architectures grows rapidly, mapping DSP programs to exploit parallelism at all levels becomes increasingly important. In this paper, we are interested in constructing efficient compile-time (static) schedules for large grain, synchronous signal processing applications [25, 35]. Large grain means that each task in the applications generates tens, or even hundreds, of outputs 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 in advance and invariable. Large-grain synchronous systems are important as they can represent a large class of DSP computations, which includes a variety of narrowband and broadband sonar systems [25, 35].

Such large-grained synchronous signal processing applications can be represented in the form of dataflow graphs, where the nodes (also called actors) represent a computation task, and arcs (or channels) represent communication between them. The communication can be viewed as a sequence of *tokens* passing through an arc. The temporal sequences of values passed along the communication channels are known as *streams*. Further, as the computation performed in a DSP application is typically on infinite sequences of values, the steams could be potentially infinite. Lee and Messerschmitt called these dataflow graphs as *synchronous* dataflow graphs, to emphasize the synchronous nature of the actors [25]. In [11, 10], these graphs are called Regular Stream Flow Graphs (RSFGs), to emphasize the abstraction mechanism of temporal sequence into streams. Henceforth we refer to large-grain synchronous systems as RSFGs[1]. Compared to the RSFGs, in the homogeneous dataflow graphs introduced by Dennis [7] the actors are fine-grain and consume/produce exactly one input/output per invocation. Further, some of the homogeneous dataflow actors, e.g. Switch and Merge are not synchronous in nature.

We are interested in constructing efficient compile-time (static) schedules as opposed to run-time (dynamic) schedules for DSP programs expressed using RSFGs. There are several distinct aspects of the scheduling problem for RSFGs.

- First, the schedule should be *balanced* — informally, this means the following. In RSFGs, the rate of consumption of tokens could be different from the rate of production. In order not to accumulate tokens infinitely on any channel, the producer and consumer need to be executed at different rates — hence the name, *multi-rate* — in the RSFG, such that the number of tokens produced and the number consumed are balanced on every channel. The proposed schedule should obey the *balanced* execution (also called firing or invocations) of the actors of RSFG. It may be observed that a *balanced* schedule requires *only* finite storage for every channel. It is well known that such a balancing condition can be ensured if the *state equations* of the RSFGs have a non-trivial solution [26, 11][2].

[1] We prefer to use the Stream Flow Graph model and its subset Regular Stream Flow Graph [11] in this paper as they have stronger verifiability compared to synchronous dataflow graphs.

[2] A formal definition of RSFG, the description of the token accumulation problem, and its relation to the state equations have been presented in Section 2.

- An RSFG may contain feedback loops which represent the precedence constraints between successive iterations. Further, these loops may be nested. The *loop-carried dependence* constraints must be obeyed in constructing schedules where the operations of successive iterations overlap. Such *software-pipelined* [2, 20] schedules are desirable as they maximize the computation rate of an iteration.

- The ultimate goal is to construct schedules which are software pipelined to achieve rate-optimality[3].

- The algorithms for the computation of the schedule should be as efficient as possible, and a low degree polynomial time complexity is desirable.

Previous work on scheduling for signal processing applications have focused only on some (but not all) of these aspects. For instance, Lee and Messerschmitt have studied the static scheduling of *synchronous dataflow graphs* (which are similar to RSFGs) and proposed a method to derive a periodic admissible schedule which does not require unbounded storage [26]. However, the schedules follow the *block* scheduling approach in which the operations of nodes in one iteration must be scheduled before the scheduling of any nodes in the next iteration. This work was extended in [40], where the study was focused on scheduling acyclic graphs. Both optimality and complexity of the scheduling methods remain as the issues. In [33], Parhi and Messerschmitt have studied the static rate-optimal scheduling of iterative dataflow flow programs via optimal unfolding. For iterative dataflow graphs, a method has been developed which can find the optimal unfolding factor and the unfolded program can be scheduled rate-optimally by exploiting successive overlap of different iterations. Chao and Sha [6] have proposed methods for constructing rate-optimal schedules for homogeneous graphs. However, these methods have focused only on homogeneous dataflow graphs. Extending their methods to RSFGs will increase the complexity of the scheduling algorithm as will be discussed in Section 7. Other related work in scheduling signal processing programs include [14, 34, 39]. A survey of related work is included in Section 7.

Like in signal processing applications, efficient loop scheduling has been a major challenge in the design of optimizing compilers for high-performance general-purpose computer architectures. *Software pipelining* has been proposed as a powerful technique in loop scheduling [2, 20]. Using this technique, the compiler derives a static parallel schedule – a periodic pattern – that overlaps instructions of a loop body from different iterations. Software pipelining has been applied to high-performance pipelined architectures, as well as super-scalar and VLIW architectures [1, 2, 8, 9, 13, 43, 20, 29, 37, 42].

The work presented in this paper is motivated by some of the insights and experience from these two research areas, namely scheduling signal processing programs and software pipelining in parallelizing compilers. We propose a novel approach for the optimal scheduling of RSFGs. Unlike the earlier approaches, our work addresses all the above mentioned aspects of the scheduling problem, and in particular the schedule is rate-optimal. Our approach to the scheduling (of RSFGs) problem can be described as follows.

[3]The notion of rate-optimality will be introduced in Section 4.

- We use a class of *linear schedules*, called *r-periodic schedules* [43], for RSFGs. A balanced schedule is obtained by making each *iteration* contain an appropriate number of instances of a node.

- We synthesize each dependence constraint between a pair of nodes in the RSFG into a set of *periodic* precedence constraints between their multiple execution *instances* in the periodic schedule. These precedence constraints are *essential* to satisfy the dependency relation. We show that such sets of constraints are linear, and hence the problem can be formulated as a linear programming problem. The linear programming formulation is referred to as the *multi-rate software pipelining (MRSP)* problem.

- From these constraints, a precedence (or dependence) graph can be constructed where all *dependence distances* are constant. Furthermore the constructed graph is *simpler* due to the fact that it does not include some of the data dependencies implied in the original RSFG as they are redundant in terms of the essential precedence relation for our purpose.

- We can solve the scheduling problem using a remarkably simple and efficient method due to Reiter [38] based on the Karp-Miller computation graphs – quite similar to our reduced precedence graphs. The key observation is that the optimal computation rate in such graphs is bounded by the maximum rate of the so-called *critical cycle(s)*. The maximal rate itself can be derived from the efficient solution of the *minimum cost-to-time ratio cycle* (MCTRC) problem [21]. The contribution of this paper is to show that they can also be used as a key step in the multi-rate scheduling of the RSFGs.

- Finally, we discuss a method based on the graph coloring algorithm to reduce the buffer space requirement for the rate-optimal schedule.

The main contribution of this paper is to demonstrate that the multi-rate software pipelining problem can be formulated as a linear programming problem. The proposed schedule is rate-optimal and fully software-pipelined. Further our solution gives the optimal unfolding factor for the graph to obtain the above rate-optimal schedule. Finally, the process of constructing the schedule has a low-degree polynomial time complexity.

The rest of this paper is organized as follows. In the following section we formally introduce RSFGs, and the balanced firings of their actors. Section 3 motivates our work with the help of two examples. Section 4 deals with formulation of the MRSP problem. In Section 5 we present an efficient solution to the MRSP problem, by relating it to the minimum cost-to-time ratio cycle (MCTRC) problem for the homogeneous precedence graph. An efficient solution for the MCTRC problem and its algorithmic complexity are also discussed in this section. In Section 6 we present our approach to reduce the buffer space requirement for the optimal schedule obtained using our linear programming framework. Our work is compared to other related work in Section 7. Finally, in Section 8 we summarize this paper and list a few possible future directions in this work.

2 Regular Stream Flow Graphs

In this section, we introduce the program model used in the rest of this paper, called the *Regular Stream Flow Graphs* (or RSFGs). In Section 2.1, we present the operational model of RSFGs. Section 2.2 introduces state equations as a tool to characterize the condition by which the tokens produced and consumed on an arc are *balanced* during an infinite firing sequence. Finally, in section 2.3, we define *timed RSFG*.

2.1 Operational Model of Regular Stream Flow Graphs

An RSFG is a digraph 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 (channels). Furthermore, associated with each input (output) arc or channel is a fixed integer n, which specifies the number of tokens consumed (or produced) on this input (output) channel for each execution of the actor. Thus, given an elementary actor A, we can therefore associate with it *a pair of vectors of integers*, $\langle \overrightarrow{In_A}, \overrightarrow{Out_A} \rangle$, the vector $\overrightarrow{In_A}$ is indexed by the input channels of A and those of $\overrightarrow{Out_A}$ are indexed by the output channels of A. If i is an input channel of A we write $\overrightarrow{In_A}[i]$ to stand for the integer corresponding to the i-indexed entry of $\overrightarrow{In_A}$. The significance of these numbers is that when the actor computes, it consumes $\overrightarrow{In_A}[i]$ tokens from each input channel i, and produces $\overrightarrow{Out_A}[o]$ tokens on each output channel o. If every component of $\overrightarrow{In_A}$ and that of $\overrightarrow{Out_A}[o]$ of a regular actor is unity then the actor is called a *homogeneous* actor.

We use the word *enabled* to mean that an actor has all that it needs to fire and *activated*, to mean that it has actually initiated a computation phase. Formally,

Definition 2.1 (Enabling Rule) *Consider an elementary actor with j input arcs $i_1, i_2, \ldots, i_j$ and k output arcs $o_1, o_2, \ldots, o_k$. The actor becomes* **enabled** *when each input arc i_m, has at least $\overrightarrow{In_A}[i_m]$ tokens, and for each output arc, o_l there are at least $\overrightarrow{Out_A}[o_l]$ empty slots on the buffer corresponding to the l-th output arc.*

The enabling rule is sometimes informally called the "firing rule".

Definition 2.2 (Activation) *An enabled actor which has begun to compute is said to be* **activated**. *During the activation, the actor will absorb $\overrightarrow{In_A}[i_m]$ tokens from the mth input buffer. It will produce $\overrightarrow{Out_A}[o_l]$ tokens, respectively, on the lth output arc. When all the output tokens are produced, the activation* **terminates**.

Without loss of generality, we make the following assumption about the way the buffers are used: 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.

Definition 2.3 (Regular Stream Flow Graph (RSFG)) *An RSFG is a graph $G = [V, E, m]$ where V is a set of* regular actors, *E is a set of directed edges representing communication channels between the actors. The* marking function *m is associated with each arc in E, and m(u,v), for every $(u, v) \in E$, is a non-negative integer representing the initial number of tokens on the arc (u, v).*

The initial number of tokens on an arc, also known as *dependency distance*, essentially characterizes possible inter-iteration dependency between the producer and the consumer. If every actor of an RSFG is homogeneous, then the graph is a special case of RSFG and we call it the *homogeneous dataflow graph*. Homogeneous dataflow graphs are widely used to represent programs in the dataflow model of computation [7].

Informally speaking, RSFGs have the same expressive power as synchronous dataflow graphs proposed by Lee and Messerschmitt [26] 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 [11]. For the purpose of this paper, we focus only on RSFGs.

2.2 State Equations

The objective of this paper is to derive rate-optimal schedules for RSFGs. One important aspect of such a schedule is that it should not require unbounded storage even under the condition that the computation may be intrinsically non-terminating. For our purpose, we are interested in a quantitative criteria for selecting an infinite firing sequence which has a bounded storage requirement, and use such criteria in our problem formulation to be presented in later sections. The formal aspects of the necessary and sufficient conditions of the existence of such schedules is beyond the scope of this paper, and can be found in [11] and elsewhere [24].

As mentioned in the introduction, the problem of unbounded buffer requirement in RSFGs is related to the *balanced* firing or *multi-rate* of firing of the actors of the RSFG. In this paper, we are only interested in infinite firing sequences that are repetitive. Let us call the repetitive pattern a *block*. Now consider an arc (u, v) in an RSFG. Intuitively, in order not to require unbounded storage for this arc, it is necessary that during the period of execution of a block, the total number of tokens produced by the node u on the channel (u, v) equals the number of tokens consumed by the node v. In this case, we say that the numbers of tokens produced and consumed are *balanced*. *State equations* are introduced below to quantitatively characterize this balancing condition.

State equations describe the state of the buffers as tokens are added to them and removed from them by the firings of actors. Each buffer is connected to exactly two actors so one can easily write a flow balance equation for each buffer in terms of the number of firings of these actors. This system of equations is called the *state equations* of the given graph. The notion of state equations is well defined for RSFGs. This is because the number of tokens produced or consumed by a regular elementary actor is known a priori. It is easy to observe that these

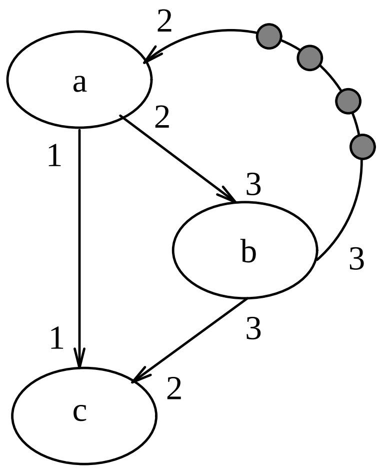


Figure 1: An Example RSFG

equations must have a solution if, in the graph, none of the arcs have token accumulation occurring during the course of a computation. Thus, in this paper, we are interested only in those RSFGs which have a solution to the state equations. Such graphs have been referred to as *well-behaved* RSFGs in [11].

In order to make the above notions clear, we present an illustrative example.

Example 2.1 Consider the data flow graph shown in Fig. 1. There are 3 actors and 4 arcs in the graph. The number of tokens produced or consumed by an actor is specified by means of a number specified close to the actor near the arc in the figure. For example, actor a produces 1 token on the arc (a, c) on each firing and consumes 2 tokens from the arc (b, a).

If the actors a, b, and c are fired $k1$, $k2$, and $k3$ times respectively, then the flow balance equations for the internal arcs of this graph are:

$$1 * k1 = 1 * k3$$

$$2 * k1 = 3 * k2$$

$$3 * k2 = 2 * k3$$

$$3 * k2 = 2 * k1$$

The above linear system of equations has a solution:

$$k1 = 3$$

$$k2 = 2$$

$$k3 = 3$$

Note that such a system may have more than one solution. In the rest of this paper, we will make extensive use of the fundamental solution — a solution in which there is at least one pair of components which are relatively prime to each other. In our example, the solution [3,2,3] is fundamental.

In this paper, we are only interested in RSFGs for which the state equations have solutions, as this is the necessary condition for the existence of a balanced firing sequence of the actors [11]. We use such solutions as a basis to derive the periodic schedules of the corresponding RSFGs.

2.3 Timed Stream Flow Graphs

Adding the notion of time to the RSFG model enables the characterization of scheduling of the systems. In this paper, we associate a deterministic time, expressed as a nonnegative real number, with each actor in an RSFG that corresponds to the execution time of the actor.

Definition 2.4 (Timed Regular Stream Flow Graphs) *A timed RSFG is a 4-tuple* $G = [V, E, m, d]$ *where* $[V, E, m]$ *represents the RSFG and* d *is a delay function such that* $d(u)$ *denotes the* **execution time** *(or the* **firing time***) of the actor* u*.*

In particular, d is a function, $d : V \rightarrow \mathcal{R}$, where V is the set of nodes in the RSFG and $\mathcal{R}$ is the set of nonnegative real numbers.

3 Motivating Example

In this section using two simple examples we will illustrate the idea of rate-optimal periodic schedules for RSFGs. Through the first example we will motivate the work reported in this paper. The second example is to illustrate the optimal unfolding of RSFGs and the concept of r-periodic schedules.

Example 3.1

Consider the RSFG shown in Fig. 1. From the discussion presented in Section 2.2, it can be observed that actors a and c need to be fired thrice for every two firings of b. Also, three firings of a, two firings of b and three firings of c constitute one iteration of the graph[4]. Let the initial number of tokens, also called the dependence distance, on the arc (b, a) be four. This intuitively means that the first two firings of a depend on the firings of b in the previous iteration. Further we assume that the execution time of each of these actors is 1 time unit and there are infinite processor resources. Lastly, throughout this paper, we will not allow two firings of an actor to take place simultaneously.

[4] We will formally define the term "iteration" in Section 4.

Time	iteration 0	iteration 1	iteration 2
0	a		
1	a		
2	b		
3	a,c		
4	b		
5	c	a	
6	c	a	
7		b	
8		a,c	
9		b	
10		c	a
11		c	a
12			b
13			a,c
14			b
15			c
16			c

Figure 2: A Rate-Optimal Schedule

A schedule for the RSFG is shown in Fig. 2. The above schedule is repetitive, starting from time step 2. Also, a new iteration is initiated after every 5 time units, or equivalently the computation rate is $\frac{1}{5}$. Thus statically compiled code of this schedule consists of a prologue, in this case the firings of the actors in the first 2 time steps (i.e., from time $t = 0$ to $t = 1$), and a repetitive pattern, from time step 2 to time step 6, in an infinite loop.[5] Observe further that in the repetitive pattern, the firings of the actors in various iterations overlap. For example, at time step 5, the firing of actor c in iteration 0 and the firing of actor a in iteration 1 take place concurrently. Such overlapped firing of actors from various iterations is known as *software pipelining* and has been used extensively in scheduling scientific programs for VLIW and superscalar processors [20, 37]. As in this case, the software-pipelined code for a loop, in general, is more efficient in terms of the computation rate of the iteration.

The schedule shown in Fig. 2 is rate-optimal in the sense that it is equivalent to the rate-optimal schedule of an equivalent homogeneous graph. We explain this idea as follows. In [10], we have established that every well-behaved stream flow graph can be emulated by an equivalent graph consisting only of homogeneous dataflow actors.[6] Since a homogeneous actor produces (or consumes) exactly one token on the output arc (or respectively from the input arc), the arcs connecting the regular actors in the stream flow graph need to be replicated depending on the number of tokens produced (or consumed) on the channel. In order to do

[5]Corresponding to the prologue, there is also an epilogue for loops with a finite number of iteration. But since our interest is on DSP applications where the computation is repeated infinitely, our schedules do not have an epilogue part.

[6]A similar claim was made by Lee et al. in [25] for *all* synchronous dataflow graphs [23]. However it was shown in [10] that such a transformation is possible if and only if the state equations of the RSFG have a non-trivial solution.

this in a consistent manner, we also need to replicate each actor a number of times equal to the number of firings of that actor as given in the state equations solution.

For the given graph, we know that actors a and c need to fire thrice while b fires only twice. Let the firings of a be named as $a0$, $a1$, and $a2$, of b as actors $b0$ and $b1$, and of c as $c0$, $c1$, and $c2$. Fig. 3 depicts the equivalent homogeneous graph. A procedure to construct an equivalent homogeneous graph is presented in [40]. As mentioned earlier, we do not allow two firings of an actor to take place concurrently. Therefore, e.g. the three instances of actor a fire sequentially. That is, the firing of $a1$ in the k-th iteration can start only after $a0$ of k-th iteration has completed its firing. This dependency, referred as the *self dependency*, is represented by means of a dotted arc from $a0$ to $a1$ in Fig. 3. Similarly there is an arc from $a1$ to $a2$. Also, $a0$ of the k-th iteration can commence its firing only after the completion of the firing of $a2$ in the $(k-1)$-th iteration. This is represented by an arc from $a2$ to $a0$ with a token on the arc. The token signifies that the dependency is from the previous iteration. Similar arcs for the instances of b and c are also shown in Fig. 3.

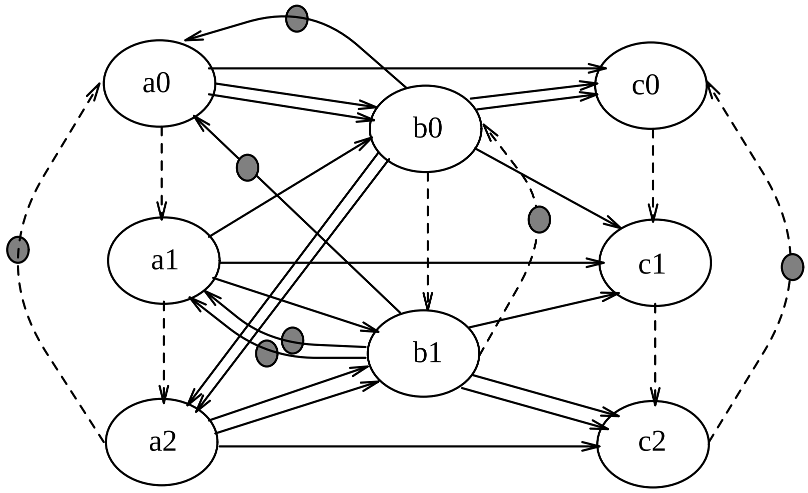


Figure 3: Equivalent Homogeneous Graph for Example 3.1

Now, we have a homogeneous graph with an initial marking. Identifying the optimal computation rate for homogeneous graphs has already been solved and reported in various forms in the literature [38, 36, 13, 30]. The following theorem characterizes the maximum rate attainable in such a graph. Proofs can be found in [38], [13] or [36].

Theorem 3.1 (Reiter [38]) *Consider a homogeneous graph $G = (V,E,m,d)$ where V is the set of nodes, E is the set of arcs, m represents the initial marking, and d the delay function of the nodes in the graph. The maximal computation rate of every node in the above graph is given by*

$$\min_{\forall C} \frac{M(C)}{D(C)}$$

where $M(C) = \sum_{(i,j)\in C} m_{ij}$, $D(C) = \sum_{i\in C} d_i$, and C is any cycle in the graph.

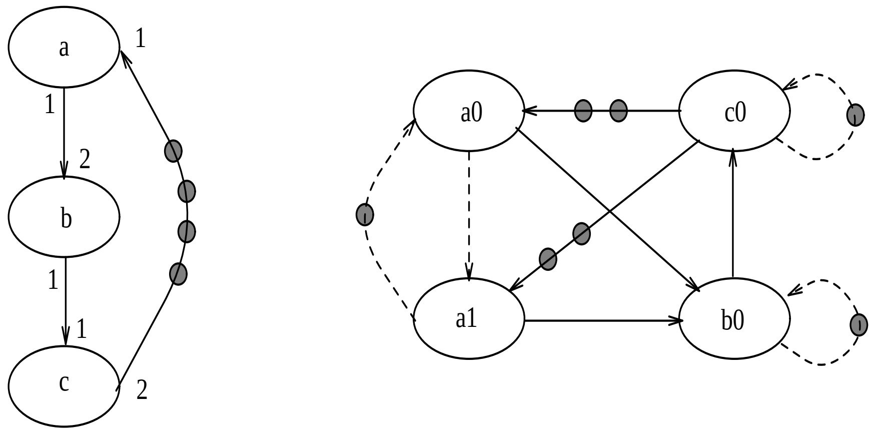
Intuitively, the constraint on the optimal computation rate is due to the cyclic dependency. The maximum computation rate $\frac{1}{T}$ is determined by the cycle, called the critical cycle, which has the smallest above ratio. It can be seen that the optimal rate for the RSFG shown in Fig. 3 is $\frac{1}{5}$, corresponding to the critical cycle

a0 $\rightarrow$ a1 $\rightarrow$ b0 $\rightarrow$ a2 $\rightarrow$ b1 $\rightarrow$ a0.

Observe that this optimal computation rate is equal to the computation rate of the schedule given in Fig. 2. Any other schedule for this RSFG must have a rate less than or equal to $\frac{1}{5}$.

Example 3.2

Next we present an example illustrating r-periodic schedules for RSFGs. Consider the graph shown in Fig. 4(a). Let the execution time of the actors a, b, and c, be respectively 1, 1, and 2 time units. Further let there be four tokens on the arc from c to a. A rate-optimal schedule for this graph is shown in Fig. 5. The symbol & is used in Fig. 5 to indicate that the execution of an actor continues in the next time step.



(a) A Regular Graph (b) Equivalent Homogeneous Graph

Figure 4: Regular Stream Flow Graph of Example 3.2

The schedule shown in Fig. 5 is periodic; but the periodicity happens once every two iterations rather than with every iteration as in the previous example. We call such a schedule as r-periodic (in this case $r = 2$). The period T for this schedule is 5, and two iterations can be initiated (one after another) in this period. This corresponds to the computation rate of $\frac{2}{5}$. To understand the significant of r-periodicity, consider once again the equivalent homogeneous dataflow graph for the RSFG (refer to Fig. 4(b)). The optimal computation rate for the homogeneous graph is $\frac{2}{5}$ corresponding to the critical cycle

$$a0 \rightarrow a1 \rightarrow b0 \rightarrow c0 \rightarrow a0.$$

The period of each iteration should be $\frac{5}{2} = 2.5$ time units. As the execution time of the actors of this graph are integers, clearly an iteration cannot be started at time $t + 2.5$ for any integer value of t. This suggests that the homogeneous graph be unfolded a suitable number of times such that any iteration commences at a time $t + T$ where t is an integer. It can be observed

Time	iteration 0	iteration 1	iteration 2	iteration 3	iteration 4	iteration 5
0	a					
1	a					
2	b	a				
3	c	a				
4	&	b				
5		c	a			
6		&	a			
7			b	a		
8			c	a		
9			&	b		
10				c	a	
11				&	a	
12					b	a
13					c	a
14					&	b
15						c
16						&

Figure 5: A Rate-Optimal Schedule

that if the homogeneous graph is unfolded twice, a schedule with an iteration period of 5 is feasible. This is what that manifests as 2-periodic schedule with an iteration period of 5 for the RSFG shown in Fig. 4(a).

As in Example 3.1, the execution of actors in various iterations do overlap in the above schedule. In fact, the extent of such overlapping, informally speaking, is more compared to that in Fig. 2. Such overlapping or software-pipelining greatly increases the computation rate. This can be observed by comparing the above schedules with a non-overlapped block schedules, proposed by Lee [26], for these graphs. The reader can verify that the block schedules for these graphs lead to a computation rate of $\frac{1}{7}$ and $\frac{1}{5}$, respectively, for the graphs shown in Fig. 1 and Fig. 4(a).

As a last remark, it may be observed that it is possible to construct the optimal schedule for a well-behaved RSFG by first transforming it into an equivalent homogeneous graph and then by identifying the critical cycle in the homogeneous graph. However, every possible initial marking of the RSFG may lead to a different equivalent homogeneous graph. For example, if the number of tokens on the arc from b to a in Fig. 1 is changed to 6, the equivalent homogeneous graph is different and is shown in Fig. 6. Further, the number of arcs in the equivalent homogeneous graph may be very large, increasing the computational complexity of the process of identifying the critical cycle. As will be shown in Appendix A, the number of edges in the equivalent homogeneous graph is given by

$$\sum_{(u,v)\in G} I_{uv} * k_v + \sum_{v\in G} k_v$$

where G is the original RSFG, I_{uv} is the number of tokens consumed by each firing of actor v on the arc (u, v), and k_v is the number of firings of v in each iteration as given by the solution

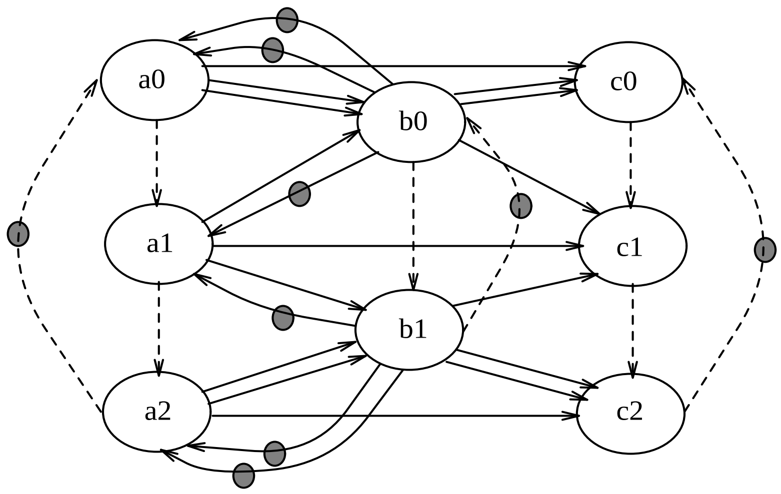


Figure 6: Equivalent Homogeneous Graph for Example 3.1 with 6 Tokens on (b, a)

to the state equations of the graph. However, it can be observed that, in the equivalent homogeneous graph, (i) some of these arcs are redundant and (ii) some of them are *inessential* in the sense that some other arc(s) in the graph capture(s) the precedence relation of the former. For example, one of the two arcs from $c0$ to $a0$ in Fig 4(b) is redundant. Also, arcs $(a0, a1)$ and $(a1, b0)$ together capture the precedence relation specified by the arc $(a0, b0)$. Thus arc $(a0, b0)$ is inessential. By removing the redundant and inessential arcs in the graph the computational complexity of constructing a rate-optimal schedule can be improved. However instead of deleting redundant and inessential arcs from the equivalent homogeneous graph, we will develop a general framework and formulate the multi-rate software pipelining problem as a linear programming problem. From the linear programming formulation, we construct a precedence graph, which consists of essential precedence relationships between the multiple instances of the actors of the RSFG. The maximal computation rate is obtained from the precedence graph by relating it to the minimum cost-to-time ratio cycle problem. The advantage of our approach compared to using the homogeneous dataflow graph is that the number of edges in the precedence graph is less than the number of edges in the homogeneous dataflow graph. As will be shown in Section 5, the number of edges in the simplified precedence graph is only

$$\sum_{(u,v)\in G} k_v + \sum_{v\in G} k_v$$

where G is the original RSFG and k_v is the number of firings of actor v in an iteration.

In the following sections, we formalize these ideas and develop the framework to construct a rate-optimal schedule for the graph of regular actors.

4 Multi-Rate Software Pipelining (MRSP): Problem Formulation

In this section we first formalize the notion of r-periodic schedule for RSFGs. Then we proceed to formulate the Multi-Rate Software Pipelining problem, henceforth referred to as the MRSP problem.

4.1 Admissible Schedule

A schedule σ is a function mapping an integer i and an actor u in the graph to a time instance t if the i-th firing of u commences at time t under the schedule σ. That is, $\sigma(i, u)$ represents the time t at which the i-th instance of actor u begins its execution. Without loss of generality, the instance number i is counted from 0 rather than from 1. Therefore the i-th instance of an actor actually means that the actor is firing for the $(i+1)$-th time. This subtle point is useful when we derive the necessary and sufficient condition for admissible schedules. If d_u represents the execution time of u, then the i-th instance of u produces output tokens at time $\sigma(i, u) + d_u$.

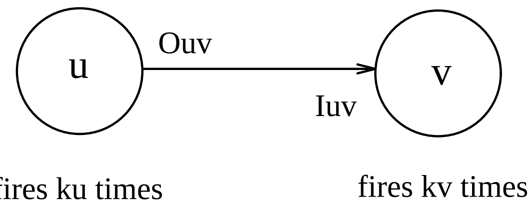


Figure 7: An Arc in the RSFG

Consider an arc (u, v) in an RSFG (refer to Fig. 7). Let u produce O_{uv} tokens in each firing and let I_{uv} be the number of tokens consumed by v in each firing. 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 4.1 *A schedule σ is said to be* **admissible** *if for every arc (u, v) in the graph, and for all $i \geq 0$,*

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

Let $X_v(t)$ denote the number of firings of actor v up to time t, including those which have commenced their execution on or before t, but not necessarily have completed the execution by time t. It is easy to observe the following properties hold:

Property 4.1 *For all $i \geq 0$, $X_v(\sigma(i, v)) = i + 1$*

Note that if the instance number is i, then it is the $(i+1)$-th firing of the actor.

Property 4.2 *For all $t \geq 0$, $\sigma((X_v(t) - 1), v) \leq t$*

Next we need to derive the number of tokens present on an arc (u, v) at time t. Let m_{uv} be the number of initial tokens on the arc (u, v). The number of firings of actor u that have completed by time t is given by $X_u(t - d_u)$. These firings produce $O_{uv} * X_u(t - d_u)$ tokens on the arc (u, v). By time t, exactly $(X_v(t) - \epsilon_v(t))$ firings of v have completed, consuming $(X_v(t) - \epsilon_v(t)) * I_{uv}$

tokens[7], where $\epsilon_v(t)$ is either 1 or 0 depending on whether or not a firing of actor v is under progress at time t. Formally,

$$\begin{aligned} \epsilon_v(t) &= 0 \qquad if \;\; \sigma(k,v)+d_v \leq t < \sigma(k+1,v), \;\; for \;\; some \;\; k \\ &= 1, \qquad otherwise \end{aligned}$$

Therefore, the number of tokens present on the arc (u, v) at time t is given by

$$b_{uv}(t) = m_{uv} + O_{uv} * X_u(t - d_u) - (X_v(t) - \epsilon_v(t)) * I_{uv}, \tag{1}$$

Now, we can establish the necessary and sufficient condition for the class of admissible schedules for RSFGs. Note that the condition somewhat resembles the characterization of the class of admissible schedules for homogeneous graphs derived in [38].

Theorem 4.1 *A schedule σ is admissible if and only if for each arc (u, v) in the graph,*

$$\sigma(i,v) \geq \sigma\left(\left\lceil \frac{(i+1)*I_{uv} - m_{uv} - O_{uv}}{O_{uv}} \right\rceil, u\right) + d_u. \tag{2}$$

Proof: [**Necessary**] First we will prove that (2) is necessary for admissible schedules. Let $\sigma(i, v) = t$. From equation (1),

$$b_{uv}(t) = m_{uv} + O_{uv} * X_u(t - d_u) - (X_v(t) - \epsilon_v(t)) * I_{uv}$$

In this case, $\epsilon_v(t) = 1$ as $\sigma(i, v) = t$. Also $X_v(t) = X_v(\sigma(i, v)) = i + 1$ by Property 4.1. Hence,

$$b_{uv}(t) = m_{uv} + O_{uv} * X_u(t - d_u) - ((i+1) - 1) * I_{uv}$$

Now, for σ to be an admissible schedule, $b_{uv}(t) \geq I_{uv}$. Therefore,

$$\begin{aligned} b_{uv}(t) = m_{uv} + O_{uv} * X_u(t-d_u) - i * I_{uv} &\geq I_{uv} \\ X_u(t-d_u) &\geq \frac{(i+1)*I_{uv} - m_{uv}}{O_{uv}} \\ X_u(t-d_u) - 1 &\geq \frac{(i+1)*I_{uv} - m_{uv}}{O_{uv}} - 1 \end{aligned}$$

Further, since $X_u(t - d_u)$ is an integer by the definition of X_u, we have

$$X_u(t-d_u) - 1 \geq \left\lceil \left(\frac{(i+1)*I_{uv} - m_{uv}}{O_{uv}} - 1 \right) \right\rceil$$

[7]Recall that our operational model assumes that the tokens present in the input arc continue to remain in the channel until the completion of the execution, at which time I_{uv} tokens are consumed by v from the channel (u, v). The results presented in this paper do not rely on this assumption since a similar proof could be given for the other extreme in which the tokens are consumed from the input channel as soon as firing of the actor commences.

Now, if we apply the σ function for actor u on either side, we get

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

Using Property 4.2,

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

Rearranging the term d_u and substituting $\sigma(i,v)$ for t, we have

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

[**Sufficient**] We have,

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

Once again, letting $\sigma(i,v)=t$ in the above equation

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

Now, applying the X_u function on either side

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

Using Property 4.1, we get

$$X_u(t-d_u) \geq \left\lceil \frac{(i+1)*I_{uv}-m_{uv}-O_{uv}}{O_{uv}}\right\rceil + 1$$

$$X_u(t-d_u) \geq \left\lceil \frac{(i+1)*I_{uv}-m_{uv}}{O_{uv}}\right\rceil$$

Also,

$$X_u(t-d_u) \geq \frac{(i+1)*I_{uv}-m_{uv}}{O_{uv}}$$

Rearranging the terms, we get

$$m_{uv} + O_{uv} * X_u(t-d_u) - i * I_{uv} \geq I_{uv}$$

This can be rewritten as

$$m_{uv} + O_{uv} * X_u(t - d_u) - ((i+1) - 1) * I_{uv} \geq I_{uv}$$

Since $\sigma(i, v) = t$, we can substitute $X_v(t)$ for $(i+1)$ and $\epsilon_v(t)$ for 1 in the above equation. Finally, using equation 1, we establish $b_{uv}(t) \geq I_{uv}$. Further, we can prove this for every v such that (u, v) is an arc in the graph. ∎

4.2 Periodic Schedules

Among the class of admissible schedules, periodic schedules are of high interest. Following the work of Reiter [38] a number of work have proposed periodic schedules for inner loops [2, 3, 8, 9, 13, 43, 20, 37] and for nested loops [12]. However, these results are applicable only for homogeneous dataflow graphs, where every actor fires exactly the same number of times. In contrast in DSP applications, as mentioned earlier it is common to have multiple rates such that the firings of the actors of an RSFG need to be *balanced.* The term *iteration* is also used to refer to the balanced firing of i the actors of an RSFG. It is defined as:

Definition 4.2 *If $[k_{v_1}, k_{v_2}, \ldots, k_{v_n}]$ is the (fundamental) solution to the state equations of a graph containing n regular actors, then an iteration of the graph consists of k_{v_i} firings of actor v_i for every $v_i \in \{v_1, v_2, \ldots, v_n\}$.*

Next, we formalize the notion of 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 4.3 *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 iteration j, $j+r$, $j+2r$, ..., take place, respectively, at time $t_{v_i,k}$, $T + t_{v_i,k}$, $2T + t_{v_i,k}$, ...*

In Example 3.1 (refer to Fig. 1), we have three actors, a, b, and c, with the solution to the state equations given by the vector $[3, 2, 3]$. In each iteration, a and c fire thrice while b fires twice. The schedule given in Fig. 2 is 1-periodic, or simply periodic, with a period $T = 5$. On the other hand, the schedule for the graph of Example 3.2 is r-periodic with $r = 2$ and $T = 5$. Observe, for example, the firings of actor b in iteration $0, 2, 4, \ldots$ take place at time steps 2, $2 + 5$, $2 + 10$, ... respectively.

Next we obtain the precedence constraints for RSFGs having r-periodic schedules.

4.3 Precedence Constraints in the RSFG

Consider an RSFG $G = (V, E, m, d)$, where $V = \{ v_1, v_2, \ldots, v_n \}$. Let $[k_{v_1}, k_{v_2}, \ldots, k_{v_n}]$ be the solution to the state equations of the graph. Using Equation 2, for every arc (u, v), such that $u, v \in \{v_1, \ldots, v_n\}$, in the graph we can write a set of equations relating the time at which the k-th instance of v fires to the time at which some k'-th firing of u as follows. Consider the k-th firing in the j-th iteration of v. Let us use the notation (j, k) 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. The (j, k)-th firing of v actually corresponds to an instance number $(j * k_v + k)$. Therefore,

$$\sigma(j, k, v) \;=\; \sigma(j * k_v + k, v)$$

Now, using this in Equation (2), we have

$$\sigma(j, k, v) \;\geq\; \sigma\left(\left\lceil \frac{(j * k_v + k + 1) * I_{uv} - m_{uv} - O_{uv}}{O_{uv}} \right\rceil, u\right) + d_u \qquad (3)$$

Equation (3) can be rewritten in the form

$$\sigma(j, k, v) \geq \sigma(j', k', u) + d_u \qquad (4)$$

Intuitively, this means that the dependency arc (u, v) in the RSFG imposes a precedence constraint between the (j, k)-th firing of v and the (j', k')-th firing of u where j' and k' represent represent respectively the iteration and the instance number in the iteration. These are given by:

$$j' = \left\lceil \frac{(j * k_v + k + 1) * I_{uv} - m_{uv} - O_{uv}}{O_{uv}} \right\rceil \;\; div \;\; k_u \qquad (5)$$

and

$$k' = \left\lceil \frac{(j * k_v + k + 1) * I_{uv} - m_{uv} - O_{uv}}{O_{uv}} \right\rceil \;\; mod \;\; k_u \qquad (6)$$

where "div" and "mod" represent respectively the integer division and the modulo operation. The expression for j' in equation (5) can be further simplified as follows.

$$j' \;=\; \left\lceil \left(\frac{j * k_v * I_{uv}}{O_{uv}} + \frac{(k+1) * I_{uv} - m_{uv} - O_{uv}}{O_{uv}} \right) \right\rceil \;\; div \;\; k_u$$

Since k_u and k_v are the number of firings of actors u and v, respectively, in each iteration, and since k_u and k_v are obtained from the state equations, it is easy to observe that $k_u * O_{uv} = k_v * I_{uv}$. Using this in the above equation and using the floor function instead of integer division, we get

$$j' \;=\; \left\lfloor \frac{1}{k_u} \left\lceil \left(j * k_u + \frac{(k+1) * I_{uv} - m_{uv} - O_{uv}}{O_{uv}} \right) \right\rceil \right\rfloor$$

As $j * k_u$ is an integer, it can be taken out of the ceiling function.

$$j' \;=\; \left\lfloor \left(j + \frac{1}{k_u} \left\lceil \frac{(k+1) * I_{uv} - m_{uv} - O_{uv}}{O_{uv}} \right\rceil \right) \right\rfloor$$

Likewise, we take j out of the floor function to get

$$j' = j + \left\lfloor \frac{1}{k_u} \left\lceil \frac{(k+1) * I_{uv} - m_{uv} - O_{uv}}{O_{uv}} \right\rceil \right\rfloor$$

Denote $\left\lfloor \frac{1}{k_u} \left\lceil \frac{(k+1)*I_{uv} - m_{uv} - O_{uv}}{O_{uv}} \right\rceil \right\rfloor$ by j_{lag}. As k lies in the range $[0, k_v)$, we can see that

$$j_{lag} = \left\lfloor \frac{1}{k_u} \left\lceil \frac{(k+1) * I_{uv} - m_{uv} - O_{uv}}{O_{uv}} \right\rceil \right\rfloor \leq 0 \tag{7}$$

Now, $j' = j + j_{lag}$ where j_{lag} is either zero or negative. Whenever j_{lag} is zero, it means that the (j, k)-th firing of v must commence d_u time units after the beginning of k'-th firing of u in the *same* iteration. However, when $j_{lag} < 0$, intuitively, there is an inter-iteration dependency between the k-th firing of v and the k'-th firing of u. The absolute value of j_{lag} gives the exact dependency distance between the firings of these two actors. Also, note that j_{lag} is independent of j, the iteration number.

The term k' in equation (6) can also be simplified as follows.

$$k' = \left\lceil \frac{(j * k_v + k + 1) * I_{uv} - m_{uv} - O_{uv}}{O_{uv}} \right\rceil \quad mod \quad k_u$$

Using the fact $k_v * I_{uv} = O_{uv} * k_u$,

$$k' = \left(j * k_u + \left\lceil \frac{(k+1) * I_{uv} - m_{uv} - O_{uv}}{O_{uv}} \right\rceil \right) \quad mod \quad k_u$$

On further simplification, we get

$$k' = \left\lceil \frac{(k+1) * I_{uv} - m_{uv} - O_{uv}}{O_{uv}} \right\rceil \quad mod \quad k_u \tag{8}$$

Equation (4) is actually a system of equations, one for each value of k in the range $[0, k_v)$. Further each corresponds to a precedence relation between an instance of u and an instance of v. For a graph containing e edges, there are essentially e such systems of equations.

Recall further that in Section 3 we have mentioned that each firing of an actor can commence only after the previous firing of that actor has completed. This can be represented by means of a self loop for each actor in the graph. That is, for every actor v we add a new arc (v, v) in the graph such that $m_{vv} = 1$, $I_{vv} = 1$, and $O_{vv} = 1$. For each of these new arcs, we write one system of equations.

$$\sigma(j, k, v) \geq \sigma(j + j_{lag}, k', v) + d_v \tag{9}$$

where

$$j_{lag} = \left\lfloor \frac{1}{k_v} \left\lceil \frac{k+1-1-1}{1} \right\rceil \right\rfloor \quad = \quad \left\lfloor \frac{(k-1)}{k_v} \right\rfloor \tag{10}$$

and

$$k' = \left\lfloor \frac{(k+1-1-1)}{1} \right\rfloor \;\; mod \;\; k_v \;\;=\;\; (k-1) \;\; mod \;\; k_v \tag{11}$$

Equation (9) is the system of equations imposing self dependency on the firing of each instance of an actor.

Our next step is to formulate the r-periodic schedule of RSFGs as an integer linear programming problem. The following subsection deals with the problem formulation.

4.4 Problem Formulation

We are interested in constructing periodic schedules for RSFGs. In particular, we will use schedules of the form $\left\lfloor \frac{j*T+A[k,v]}{r} \right\rfloor$ for the (j,k)-th firing of actor v. That is,

$$\sigma(j,k,v) = \left\lfloor \frac{j*T+A[k,v]}{r} \right\rfloor, \tag{12}$$

where $A[k,v] \geq 0$ is a constant. A similar form of linear schedules has been used for scheduling instructions in an innermost loop in [43]. To find a schedule of the above form, we first consider, as in [38], the use of the rational schedule

$$\sigma^*(j,k,v) = \frac{j*T+A[k,v]}{r} \tag{13}$$

It has been shown in [43] that a schedule σ is valid if its rational form σ^* is valid. Therefore, using the rational form, the equation (4) can be rewritten as

$$\frac{j*T+A[k,v]}{r} \;\; \geq$$
$$\frac{\left(j+\left\lfloor \frac{1}{k_u}\left\lceil \frac{(k+1)*I_{uv}-m_{uv}-O_{uv}}{O_{uv}} \right\rceil \right\rfloor\right)*T + A\left[\left(\left\lceil \frac{(k+1)*I_{uv}-m_{uv}-O_{uv}}{O_{uv}} \right\rceil \;\; mod \;\; k_u\right), u\right]}{r} + d_u \tag{14}$$

Simplifying (14) further, we get

$$A[k,v] - A\left[\left(\left\lceil \frac{(k+1)*I_{uv}-m_{uv}-O_{uv}}{O_{uv}} \right\rceil \;\; mod \;\; k_u\right), u\right] \;\; \geq$$
$$T*\left\lfloor \frac{1}{k_u}\left\lceil \frac{(k+1)*I_{uv}-m_{uv}-O_{uv}}{O_{uv}} \right\rceil \right\rfloor + r*d_u \tag{15}$$

Note that Equation (15) is independent of the iteration number j. Also, the above equation is in the linear form

$$A[k,v] - A[k',u] \;\geq\; T*j_{lag} + r*d_u,$$

where k' and j_{lag} are defined by Equations (8) and (7). Thus, we can say that (15) forms a system of linear inequalities, for each value of $k \in [0, k_v)$, In a similar manner, substituting (13) in the system of equations which impose self dependency (Equation (9)), we get

$$\frac{j * T + A[k, v]}{r} \geq \frac{\left(j + \left\lfloor \frac{(k-1)}{k_v} \right\rfloor\right) * T + A\left[((k-1) \ \ mod \ \ k_v), v\right]}{r} + d_v \tag{16}$$

This can be simplified further as,

$$A[k, v] - A[((k-1) \ \ mod \ \ k_v), v] \geq T * \left\lfloor \frac{(k-1)}{k_v} \right\rfloor + r * d_v \tag{17}$$

Using the systems of equations (15) and (17), the MRSP problem can be formulated as:

[MRSP Problem] minimize $\frac{T}{r}$ subject to the conditions

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

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

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

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

In the following subsection we will obtain the MRSP formulation for our motivating example.

4.5 Back to Motivating Examples

In the RSFG shown in Fig. 1, there are four arcs $(a, b), (b, a), (a, c)$, and (b, c). The solution to the state equations of the graph is given by $[3, 2, 3]$. First we calculate the values of j_{lag} and k' for these four arcs. The values are shown in the following table.

Arc	Parameter	k = 0	k = 1	k = 2
(a,b)	$j_{lag} = \lfloor \frac{1}{3} \lceil \frac{3*(k+1)-0-2}{2} \rceil \rfloor$	0	0	–
	$k' = \lceil \frac{3*(k+1)-0-2}{2} \rceil \;\; mod \;\; 3$	1	2	–
(b,a)	$j_{lag} = \lfloor \frac{1}{2} \lceil \frac{2*(k+1)-4-3}{3} \rceil \rfloor$	-1	-1	0
	$k' = \lceil \frac{2*(k+1)-4-3}{3} \rceil \;\; mod \;\; 2$	1	1	0
(b,c)	$j_{lag} = \lfloor \frac{1}{2} \lceil \frac{2*(k+1)-0-3}{3} \rceil \rfloor$	0	0	0
	$k' = \lceil \frac{2*(k+1)-0-3}{3} \rceil \;\; mod \;\; 2$	0	1	1
(a,c)	$j_{lag} = \lfloor \frac{1}{3} \lceil \frac{1*(k+1)-0-1}{1} \rceil \rfloor$	0	0	0
	$k' = \lceil \frac{1*(k+1)-0-1}{1} \rceil \;\; mod \;\; 3$	0	1	2

We have assumed that the execution time of each actor is 1 time unit, i.e. $d_a = d_b = d_c = 1$. Using the values of k' and j_{lag} from the above table, Constraint I in the MRSP formulation leads to the following inequalities.

$$\begin{aligned}
A[0,b] - A[1,a] &\geq r \\
A[1,b] - A[2,a] &\geq r \\
A[0,a] - A[1,b] &\geq -T + r \\
A[1,a] - A[1,b] &\geq -T + r \\
A[2,a] - A[0,b] &\geq r \\
A[0,c] - A[0,b] &\geq r \\
A[1,c] - A[1,b] &\geq r \\
A[2,c] - A[1,b] &\geq r \\
A[0,c] - A[0,a] &\geq r \\
A[1,c] - A[1,a] &\geq r \\
A[2,c] - A[2,a] &\geq r
\end{aligned} \tag{18}$$

In a similar way, it can be seen that Constraint II in the MRSP problem leads to the following inequalities.

$$\begin{aligned}
A[0,a] - A[2,a] &\geq -T + r \\
A[1,a] - A[0,a] &\geq r \\
A[2,a] - A[1,a] &\geq r \\
A[0,b] - A[1,b] &\geq -T + r \\
A[1,b] - A[0,b] &\geq r \\
A[0,c] - A[2,c] &\geq -T + r \\
A[1,c] - A[0,c] &\geq r \\
A[2,c] - A[1,c] &\geq r
\end{aligned} \tag{19}$$

Thus, the MRSP formulation for the RSFG shown in Fig. 1 is to minimize $\frac{T}{r}$ subject to the equations (18) and (19).

The precedence constraints for the RSFG shown in Fig. 4 can be obtained in a similar way. In the following section, we present an efficient solution for the MRSP problem.

5 Solution to the MRSP Problem

The MRSP problem formulated in the previous section is actually a linear programming problem which could be solved using classical linear programming methods such as the simplex method. However, an efficient solution for this problem can be obtained by identifying the MRSP problem as a particular application of the well-known *Minimum Cost-to-Time Ratio Cycle (MCTRC)* problem in integer linear programming. In order to do this, first we construct a homogeneous precedence graph, or simply the precedence graph, from the system of constraints (constraints (I) and (II) in the MRSP problem formulation). In Section 5.1, we first describe a procedure to construct the precedence graphs. An efficient solution to determine the maximal computation rate of the RSFG is presented in Section 5.2. Using the maximal computation rate, a rate-optimal schedule can be constructed. We also discuss the complexity of the construction procedure in this section.

5.1 Construction of Equivalent Precedence Graph

Constraints (I) and (II) in the MRSP problem essentially specify precedence relations between the firings of the various instances of the actors of the RSFG. We will represent the set of precedence relations by a graph, called the *precedence graph.* The precedence graph consists of multiple instances of each actor in the RSFG, the number of occurrences being specified by the solution to the state equations. Each constraint obtained from constraints I and II in the MRSP problem is represented by means of an edge in the precedence graph. As the absolute value of j_{lag} in the constraints represent the dependency distance, as mentioned in Section 4.3, it corresponds to the initial tokens on the precedence graphs. Since in the precedence graph, as the name indicates, arcs represent only precedence relations, the actors are homogeneous. Lastly, we can observe that the precedence graph is similar to the Karp-Miller computation graphs [38].

Consider the graph of regular actors $G = (V, E, m, d)$. The homogeneous precedence graph for G consists of one node for each instance of actor of G in an iteration. That is, for every node $v_i \in V$, there are k_{v_i} nodes in the precedence graph. Let us name these nodes as $v_i^0,\ v_i^1,\ \ldots\ ,\ v_i^{k_{v_i}-1}$. Further, for every edge $e = (v_j, v_i) \in E$, we have a system of equations as in (4). In this system of equations, for each value of $k \in [0, k_{v_i})$ we have an equation representing a dependency relation from some $v_j^{k'}$ to v_i^k. Each of these equations is represented by means of a directed edge from $v_j^{k'}$ to v_i^k in the precedence graph. The initial number of tokens present on this arc is equal to the absolute value of j_{lag}. Thus in the precedence graph, we only include one essential arc corresponding to each arc in the original RSFG. Further, Equation (9) imposes additional dependency constraints on the execution of two instances of an actor v_i.

It is easy to observe that there is a dependency from v_i^k to $v_i^{(k+1)}$, for each $k \in [0, (k_{v_i} - 2)]$ with no initial token. Further, there is a dependency relation from $v_i^{(k_{v_i}-1)}$ to v_i^0 and this arc contains exactly one initial token. This is true for each actor $v_i \in V$. The precedence graph constructed in this way is a network consisting only *homogeneous* actors. We will represent the homogeneous precedence graph as a 4-tuple $(\mathcal{V}, \mathcal{E}, \mathcal{M}, \mathcal{D})$, where $\mathcal{V}, \mathcal{E}, \mathcal{M}$, and $\mathcal{D}$ represent respectively the nodes, edges, initial tokens on the edges, and the delay function with each node in the precedence graph.

Algorithm 5.1

[Input] A regular graph $G = (V, E, m, d)$, and the solution vector $[k_{v_1},\ k_{v_2},\ \ldots,\ k_{v_n}]$ for the state equations of G
[Output] The homogeneous precedence graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{M}, \mathcal{D})$

Step 1 : Initialize $\mathcal{V} = \phi$, $\mathcal{E} = \phi$, $\mathcal{M} = \phi$, and $\mathcal{D} = \phi$, where ϕ represents the empty set.

Step 2 : For each $v_i \in V$ do

- **Step 2.1** : add nodes $v_i^0,\ \ldots,\ v_i^{(k_{v_i}-1)}$ to the set $\mathcal{V}$.
- **Step 2.2** : derive the precedence constraint imposed by the self dependency, using Equation (9).
- **Step 2.3** : For each $k \in [0, k_{v_i})$ do
  - **Step 2.3.1** : Compute the values of k', j_{lag}
  - **Step 2.3.2** : Include an edge $(v_i^{k'}, v_i^k)$ in the set $\mathcal{E}$
  - **Step 2.3.3** : Set $\mathcal{M}_{(v_i^{k'}, v_i^k)} = abs(j_{lag})$
  - **Step 2.3.4** : Set $\mathcal{D}_{v_i^k} = d_{v_i}$

Step 3 : For each $(v_j, v_i) \in E$ do

- **Step 3.1** : derive the precedence constraint due to this arc using Equation (4).
- **Step 3.2** : For each $k \in [0, k_{v_i})$ do
  - **Step 3.2.1** : Compute the values of k', j_{lag}
  - **Step 3.2.2** : Include an edge $(v_j^{k'}, v_i^k)$ in the set $\mathcal{E}$
  - **Step 3.2.3** : Set $\mathcal{M}_{(v_j^{k'}, v_i^k)} = abs(j_{lag})$

We will quickly derive the complexity of this algorithm. Let us assume the number of nodes and the number of edges in the RSFG be of the order n and m respectively. Further, let k represent the average of the number of firings of each actor as specified by the solution to the state equations. It can be seen that Step 2.3 iterates for $\mathcal{O}(k)$ times while Step 2 itself is repeated $\mathcal{O}(n)$. Thus Step 2 to Step 2.3.4 has a complexity $\mathcal{O}(kn)$. Similarly Step 3 to Step 3.2.3 has a time complexity $\mathcal{O}(mk)$. Thus the above construction algorithm has a time complexity $\mathcal{O}((m + n)k)$. Usually m is also of $\mathcal{O}(n)$, and therefore the complexity is $\mathcal{O}(nk)$.

Next we calculate the number of nodes and the number of edges in the precedence graph. It can be easily seen (from Step 2.1) that the precedence graph for G consists of $\sum_{v_i \in V} k_{v_i}$ nodes. Step 2.3.2 adds k_{v_i} edges to the precedence graph for each v_i. Thus there are $\sum_{v_i \in V} k_{v_i}$ such arcs, referred to as the self dependency arcs. Further, Step 3.2 contributes k_{v_i} edges for each edge (v_j, v_i) in the original RSFG. There will be a total of $\sum_{(v_j,v_i) \in E} k_{v_i}$ such edges in the precedence graph. Thus the total number of edges in the precedence graph is given by

$$\sum_{v_i \in V} k_{v_i} + \sum_{(v_j,v_i) \in E} k_{v_i}.$$

In the above order notation, the number of edges in the precedence graph is $\mathcal{O}(nk)$. Let us compare the number of edges in the precedence graph with that in the equivalent homogeneous graph discussed in Section 3. As explained in Appendix A, the number of edges in the homogeneous graph is given by

$$\sum_{v_i \in V} k_{v_i} + \sum_{(v_j,v_i) \in E} I_{uv} \times k_{v_i}.$$

Precedence Graphs for the Examples

Now, we will construct the precedence graph for the RSFGs shown in Fig. 1 and Fig. 4. For the RSFG of Example 3.1, the precedence constraints are given by Equations (18) and (19) in Section 4.5. The precedence graph constructed using these equations is shown in Fig. 8(a). Likewise, the precedence graph for the RSFG of Example 3.2 is constructed and is shown in Fig. 8(b). Observe that the precedence graphs in Fig. 8 are similar to, but much simpler than, the equivalent homogeneous dataflow graphs shown in Fig. 3 and Fig. 4(b). In particular, the number of edges in the precedence graph is much less compared to that in the homogeneous graph. This is because, the precedence graph captures only the essential precedence relation for every data dependency arc in the original RSFG. For example the arc (a, b) in Fig. 1 is captured by means of 6 arcs in the equivalent homogeneous graph (refer to Fig. 3) while only 3 precedence relations are required in Fig. 8(a).

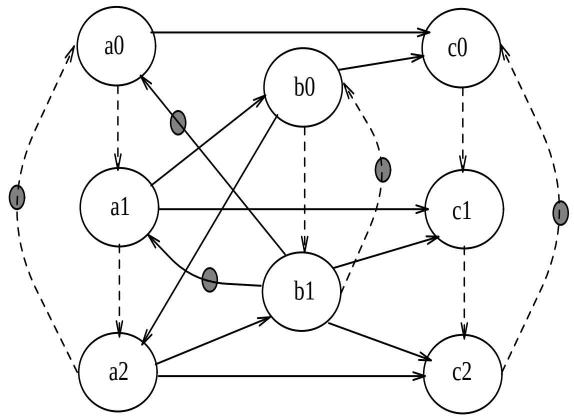
5.2 An Efficient Algorithm for MCTRC

The precedence graph constructed as mentioned in the previous subsection captures the precedence constraints specified in the MRSP problem. Thus an optimal computation for the original RSFG is equivalent is equal to the optimal computation rate attainable in the homogeneous precedence graph. From Theorem 3.1, the maximal computation rate in the homogeneous precedence graph is

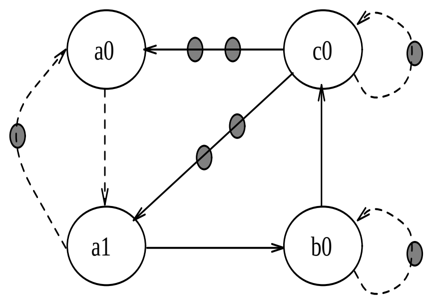
$$\frac{r}{T} = min_{\ \forall \mathcal{C}} \ \frac{\sum_{\forall\ \varepsilon\ \in\ \mathcal{C}} \ m_{\varepsilon}}{\sum_{\forall\ \nu\ \in\ \mathcal{C}} \ d_{\nu}}, \tag{20}$$

where $\mathcal{C}$ is any cycle in the precedence graph.

Computing the optimal rate in the homogeneous graph can be related to the MCTRC problem in linear programming. Karp's algorithm [19] to compute the minimum cycle mean



(a) Precedence Graph for Example 3.1



(b) Precedence Graph for Example 3.2

Figure 8: Homogeneous Precedence Graphs

is directly related to the MCTRC problem and has a complexity of $\mathcal{O}(ln)$, where l is number of edges and n is the number of nodes in the precedence graph. As mentioned in Section 5.1, the precedence graph has $\mathcal{O}(nk)$ edges and $\mathcal{O}(nk)$ nodes. Thus the complexity is $\mathcal{O}((nk)^2)$ for the RSFGs. The MCTRC problem can also be solved by using a binary search to find the appropriate value of T/r. In doing so, $\mathcal{O}(\lg(nk))$ particular shortest path problems are to be solved. The complexity of the shortest path algorithm is $\mathcal{O}((nk)^2)$ for the Bellman-Ford algorithm or $\mathcal{O}((nk)^{\frac{3}{2}} \ \lg(nk))$ for the Gabow-Tarjan algorithm [21]. Thus, the overall complexity of the MCTRC problem using these approaches are $\mathcal{O}((nk)^2 \ \lg\ (nk))$, when using the Bellman-Ford algorithm, and $\mathcal{O}((nk)^{\frac{3}{2}} \ \lg^2(nk))$ for Gabow-Tarjan algorithm.

Note that once the value of T/r is known, we need to determine the values of $A[k,v]$'s subject to the condition given by equations (15) and (17). These are

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

and

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

Once again, for various values of k in the range $[0, k_v)$ and for various (u,v), we get a number such constraints. The values of $A[k,v]$ for all v and for all k in the appropriate range, must satisfy these constraints. As explained in [21], this problem is a particular shortest path problem, and it can be solved in $\mathcal{O}((nk)^2)$ time with the Bellman-Ford algorithm and in $\mathcal{O}((nk)^{\frac{3}{2}} \ \lg(nk))$ time with the Gabow-Tarjan algorithm. Thus the overall complexity of the MRSP problem is $\mathcal{O}((nk)^2)$ using Karp's algorithm, $\mathcal{O}((nk)^2 + (nk)^2 \ \lg\ (nk))$ using the Bellman-Ford algorithm, or $\mathcal{O}((nk)^{\frac{3}{2}} \ lg^2(nk) + (nk)^{\frac{3}{2}} \ lg(nk))$ using the Gabow-Tarjan algorithm. Clearly, the overall complexity of our solution to the MRSP problem is in a low degree polynomial.

Optimal Computation Rates for the Examples

Let us compute optimal rate for Example 3.1 using the precedence graph and Theorem 3.1. Since the execution time of actors a, b, and c in the original RSFG is unity, the execution time of all actors in the precedence graph is also unity. It can be seen that the path $a0 \rightarrow a1 \rightarrow bo \rightarrow a2 \rightarrow b1 \rightarrow a0$ is the critical path in the precedence graph. The maximum computation rate for this cycle is $\frac{1}{5}$. Now using this computation rate, that is $\frac{r}{T} = \frac{1}{5}$, we need to solve the equations specified by Constraints (I) and (II) of the MRSP problem to obtain the values for various $A[k,v]$'s. It can be seen that the solution $A[0,a] = 0, A[1,a] = 1, A[2,a] = 3, A[0,b] = 2, A[1,b] = 4, A[0,c] = 3, A[1,c] = 5, i$ and $A[2,c] = 6$ satisfies the constraints and corresponds to the schedule depicted in Fig. 2. ∎

Consider the RSFG shown in Fig. 4(a). The precedence graph for this example is shown in Fig. 8(b). The execution time of the homogeneous actors $a0$, $a1$, and $b0$ is unity and while that of $c0$ is 2 time units. The critical cycle in the precedence graph is

$$a0 \rightarrow a1 \rightarrow b0 \rightarrow c0 \rightarrow a0.$$

The computation rate for the graph is $\frac{2}{1+1+1+2} = \frac{2}{5}$. Solving Constraints (I) and (II) of the MRSP problem with this computation rate, we get, $A[0,a] = 0, A[1,a] = 2, A[0,b] = 4$, and $A[0,c] = 6$, which corresponds to the schedule shown in Fig. 5. ∎

6 An Approach to Minimize Buffer Space

In this section we present our approach to minimize the buffer space requirement for the rate-optimal schedule. Our approach is based on a two-phase method. In the first phase, we compute the buffer space requirement for each arc in an equivalent homogeneous data dependency graph. Using the buffer requirements for the arcs in homogeneous graph, the buffer requirements for the arcs of the RSFG are calculated. In the second phase, the buffer requirements for the arcs of the original RSFG are further reduced using a graph coloring algorithm. A key observation here is that since the schedule is already fixed (as described in Sections 4 and 5), the problem of minimizing the buffer space can be formulated as a graph coloring problem, similar to the conventional register allocation problem [4, 5, 15]. In this paper we use the coloring method proposed in [15] which is based on a novel cyclic interval graph representation.

We illustrate our approach using the motivating example considered in Section 3. For the buffer allocation problem, we consider the equivalent homogeneous data dependency graph, rather than the precedence graph. This is because in the buffer allocation problem one needs to consider the actual data movements between the actors, which is explicitly represented in the homogeneous data dependency graph. In contrast, in the precedence graph the producer-consumer relationship between actors is not explicit. In particular, the producer-consumer relation between some pairs of actors may have been omitted. For example, in the precedence graph of Example 3.1 (refer to Fig. 8(a)), the producer-consumer relation between actors $a0$ and $b0$ has been completely suppressed. Further, we do not consider self-dependency arcs in the homogeneous graph (represented using dotted lines in Fig. 3) as these are merely for the scheduling purposes.

Phase 1 – Buffer Allocation

In the first phase we compute the buffer space required for each arc in the homogeneous data dependency graph. For the RSFG used in Example 3.1, the homogeneous graph is shown in Fig. 3. Note that each arc in the original RSFG manifests itself into a number of arcs in the homogeneous dependency graph. In particular, the arcs (b, a) and (a, b) in the RSFG manifests as 12 different arcs, numbered from $e1$ to $e12$, in the homogeneous graph. The buffer space required for an arc in the homogeneous dependency graph may be greater than 1. This is because the actors are executed asynchronously, and as a consequence, an actor may fire a number of times even before one of its successor actors is fired once. The buffer space required for each arc in the homogeneous graph can be computed in a number of ways (refer, for example, to [32]). For the homogeneous graph shown in Fig. 3, the buffer space associated with each arc is 1. Further the buffers assigned for an arc can be reused in the successive execution of the repetitive schedule. Once the buffer sizes have been computed for various arcs in the homogeneous graph, the buffer sizes of the arcs which correspond to a single arc in the original RSFG can be grouped together and thus the size of buffer for the original arc can be determined. For example, grouping arcs $e1$ through $e6$, we get the buffer space required for the arc (b, a) in the original RSFG. A buffer size of 6 is required for each of the arcs (a, b) and (b, c) in the RSFG.

It can be seen that the rate-optimal repetitive schedule can be supported using a buffer size of 6 each for the two arcs (a, b) and (b, c). However the buffer sizes thus computed may not be minimal. For example, for the arc (a, b) in the original RSFG, one can verify that a buffer size of 4 is sufficient to support the rate-optimal schedule. A tight bound on the buffer space requirement can be obtained using the graph coloring method on the cyclic interval graph [15]. This is done in the second phase of our algorithm.

Phase 2 – Graph Coloring

In conventional graph coloring algorithms, the live ranges of each program variable is represented using either an interference graph [4] or a cyclic-interval graph [15]. Based on the overlapping of the live-ranges of the program variables registers are shared by the variables: if the live-ranges of two variables do not overlap, then they can share the same register.

In our case since each buffer location corresponds to a program variable (at least conceptually), we need to consider the live ranges of the values written in each of these buffer locations. Wherever the live ranges do not overlap, buffer spaces could be shared. In this paper, we restrict our attention to sharing buffer spaces among arcs (in the homogeneous dependency graph) which are manifestations of the *same* arc in the original RSFG[8]. Thus, we will only consider sharing buffer space amongs arcs $e1$ through $e6$ (refer to Fig. 9), which are manifestations of arc (b, a) in the RSFG or amongs arcs $e7$ through $e12$, which are manifestations of arc (a, b) in the RSFG .

[8]Sharing buffer space among all arcs in the homogeneous graph is more involved. Refer to [31] for a discussion on this.

The live ranges of the buffer locations for arcs $e1$ through $e12$ are shown in Fig. 9. First, let us explain the notation used in this figure. A point of time at which a token is produced by an actor (or the value carried by the token in the written in the associated output arc) is called the *definition point* and is represented using a black dot. The *last use point* of a token is the time at which the token is consumed by the consumer. The last use point is represented using an arrowhead. If the live-range of a variable goes across iterations, it is represented by means of a dotted cyclic arc across the iteration. (Refer to [15] for further details on the hierarchical cyclic interval graph representation.) The lives ranges of the arcs of the homogeneous graphs are shown in Fig. 9. For brevity, we have considered only the arcs (b, a) and (a, b) of the original RSFG.

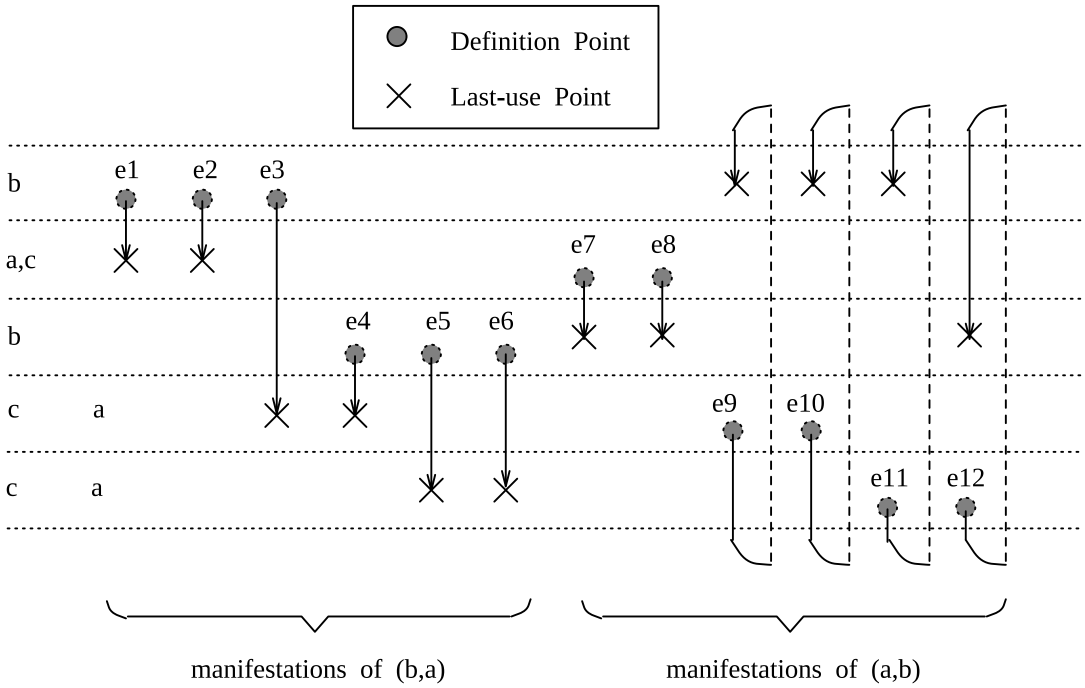


Figure 9: Live Ranges of Arcs of the Homogeneous Graph

As can be seen from Fig. 9, the live ranges of buffer spaces for arcs $e1$ and $e2$ do not overlap with the live ranges of arcs $e4$, $e5$, or $e6$. Thus, for example, the buffer space for $e1$ and $e2$ can be shared by arcs $e4$ and $e5$, respectively. The live range of $e3$ on the other hand overlaps with the live ranges of the arcs $e1, e2, e4, e5$, and $e6$ and hence its buffer space cannot be shared by any other arc. Thus, effectively, we need to allocate 1 buffer space for arcs $e1$, $e2$, $e3$, and $e6$. Arcs $e4$ and $e5$ will share the buffer space provided for arcs $e1$ and $e2$ respectively. Now, grouping the arcs $e1$ through $e6$, we get the buffer space requirement for arc (b, a) in the original RSFG. On buffer sharing, the buffer space requirement reduces to 4. In a similar way, arcs $e7$ and $e8$ can share their buffer space with arcs $e9$ and $e10$ respectively.

Presently the coloring algorithm based on the cyclic interval graph representation has been implemented for homogeneous graphs [31]. We plan to extend this tool for reducing buffer space requirement for RSFGs.

7 Related Work

As mentioned earlier this paper ties together research results in two related but distinct areas: (1) scheduling in application-specific architectures for applications such as DSP, and (2) loop scheduling in optimizing/parallelizing compilers for general-purpose architectures. In this section, we discuss the related work in both areas.

7.1 Scheduling in DSP Architectures and Applications

The stream flow graph model used in this paper was partly inspired by several program representations for signal processing, most notably the *synchronous dataflow graphs* [26], as well as stream based dataflow programming languages, such as Silage [16] and Gabriel [22].

In [26], Lee has studied the static scheduling of synchronous dataflow programs and proposed a method to derive a periodic admissible schedule which does not require unbounded storage. Sih and Lee have studied the scheduling of acyclic synchronous dataflow graphs, and proposed several heuristic solution methods [40]. Compile-time scheduling of dataflow program graphs with dynamic constructs was reported in [14]. The multiprocessor scheduling and program mapping have been studied in [40] and [35]. Except for [40] which deals with acyclic graphs, all other work follow the *block* scheduling approach.

In the block scheduling approach, the execution of nodes in one iteration must be completed before the execution of any nodes in the subsequent iteration, i.e. the operations from consecutive iterations do not overlap. Hence, such schedules are called non-overlapping schedules. Further, in the block scheduling method [26], a compiler emulates the execution of the graph under an operational model. In this sense, it is similar to the software pipelining method proposed in [2, 13]. An optimal non-overlapped schedule using the block scheduling approach[9] leads to a computation rate of $\frac{1}{7}$ for the graph shown in Fig. 1, and $\frac{1}{5}$ for the graph in Fig. 4(a). In comparison, our schedules (refer to Fig. 2 and Fig. 5) have an optimal computation rate of $\frac{1}{5}$ and $\frac{2}{5}$ for these two graphs. It may be possible in certain cases to *retime* (also known as the 'cut-set transformation') [27] the dataflow graph to achieve rate-optimal non-overlapped schedules. However, as pointed out by Parhi and Messerschmitt [33], constructing rate-optimal non-overlapped schedules using retiming cannot *always* be guaranteed. Thus the block scheduling approach cannot guarantee rate-optimal schedules in all cases. In particular, retiming cannot ensure rate optimality for non-overlapped schedules in cases where the graph needs to be unfolded a number of times. The graph shown in Fig. 4(a) is an example. Cubric and Panangaden [44] have proposed algorithms to construct schedules which achieve either minimal buffer requirement or maximal concurrency independently for regular stream flow graphs. Their approach is based on an operational model and results in non-overlapped schedules.

In [33], Parhi and Messerschmitt have studied the static rate-optimal scheduling of iterative dataflow flow programs via optimal unfolding. For iterative dataflow graphs, a method has been developed which can find the optimal unfolding factor and the resulting unfolded program

[9] These schedules are shown in Appendix B.

can be scheduled rate-optimally. However, no algorithm is presented in [33] to construct overlapped rate-optimal periodic schedule. Further, the method has focused *only* on homogeneous dataflow graphs. Recently, Lucke and Parhi [28] 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 [6] 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. Thiele [41] work on constructing resource-constrained rate-optimal schedule for systolic is also based on homogeneous dataflow graphs. Even though Parhi's, Thiele's or Chao's method can be applied to RSFGs by first converting the RSFG into an equivalent homogeneous graph, the computational complexity of such an approach would be much greater than that of MRSP. This is because, the number of edges in the homogeneous dataflow graph is larger than that in the precedence graph by roughly a factor of $\mathcal{I}$, where $\mathcal{I}$ is average of the various I_{uv}'s. Further, our method to compute the unfolding factor r, does not rely on enumerating all the cycles in the graph. In fact, for scheduling purposes, we only need to know the ratio T/r, which can be obtained from the solution to the MCTRC problem.

Printz [35] discusses a method to map large-grain synchronous signal processing programs onto multiprocessor machines. In this method, signal processing applications are expressed using a style which combines serial, systolic, and data-parallel implementations. Further, a scheduling algorithm based on list-scheduling heuristics has been proposed. The scheduling algorithm also considers inter-processor communication costs. The scheduling and mapping obtained using this method are not optimal, though it is guaranteed that schedule is within a certain factor from the optimality. A complete DSP design environment, called McDAS was developed by Hoang [17]. In particular, McDAS implements a scheduler which considers inter-processor communication delays and resource availability. Further, the scheduling algorithm follows an *iterative clustering* approach; the iterative refinement allows the algorithm to improve on the last scheduling step while clustering combines actors with heavy communication costs together to reduce inter-processor communication delays. Related work applied in the area of high-level synthesis and code generation can be found in [39, 34].

7.2 Loop Scheduling in Parallelizing Compilers

The efficient exploitation of fine-grain parallelism in loops has been a major challenge in the design of optimizing compilers for high-performance computer architectures. Fine-grain loop scheduling involves the detection of parallelism from a partially ordered set of operations of the loop body which are to be performed repetitively over a sequence of iterations. With loop-carried dependencies, the dependency relation between operations in the loop is a digraph with cycles, thus presenting a specific challenge to the effectiveness of classical scheduling theory. *Software pipelining* has been proposed as a technique for fine-grain loop scheduling. This method derives a static parallel schedule – a periodic pattern – that overlaps instructions of a loop body from different iterations. Software pipelining has been proposed for high-performance pipelined processor architectures, as well as super-scalar and VLIW architectures [1, 2, 8, 9, 20, 29, 37, 42].

There are essentially two approaches for software pipelining. In the first approach [2, 13], a compiler emulates the execution of the loop under a certain operational model, until a periodic pattern is "discovered" which often consists of instructions from a number of consecutive iterations. In the technique proposed in [2], the instructions are scheduled as early as possible by a *greedy* machine model with the constraint of having *bounded gaps* between the computations. In [13], the loop is modeled by a timed Petri net executed under the "earliest firing rule" to find the repetitive pattern.

The second approach generates a fixed scheduling pattern for one iteration, and repeats it for all iterations with a specific initiation time interval. In [20], the author gives a mathematical (linear programming) formulation of the scheduling problem, and tries to minimize the initiation interval between the successive iterations. However, the requirement that only regular schedules (such that all iterations are scheduled in the same manner) is considered may exclude some optimal solutions. This was demonstrated in [18], and an improvement was suggested in which the loops may first be subjected to an *unrolling* transformation.

In [43], an alternative method has been proposed, based on an elegant and powerful periodic scheduling technique for Karp-Miller computation graphs [38]. The formulation also does not have the restriction that the schedule of each individual iteration must be the same. In fact, the schedule of two consecutive iterations may not be the same, but will repeat every r iterations; hence called *r-periodic*. A polynomial time algorithm to find the *optimal schedule* that maximizes the computation rate has been presented in [43]. Most of these work on software pipelining have focused on a single loop (or the innermost loop in a loop nest). For work on software pipelining for nested loops, readers are referred to [12].

8 Conclusions and Future Work

In this paper we have studied the problem of constructing rate-optimal schedules for a class of dataflow graphs called regular stream flow graphs. These graphs are useful to express DSP applications in which the firing rates of the actors of the graph may not be identical. The problem of rate-optimal scheduling has been formulated as a linear programming problem and an efficient solution for this problem has been proposed by relating it to the well-known minimum cost-to-time ration problem in linear programming. An algorithm, which has a low degree polynomial complexity, has been presented to construct the rate-optimal schedule. 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. We have proposed a two-phase method to reduce the buffer space requirements for the optimal schedule. For the rate-optimal schedule, a graph coloring algorithm based on the new cyclic-interval graph representation has been discussed to achieve minimal buffer space requirements for the arcs of the RSFG

In [31], a linear programming framework has been proposed to perform concurrent rate-optimal scheduling and register allocation for homogeneous graphs. In this method, among the possible rate-optimal schedules, the one that uses minimum buffer space is identified using a linear programming formulation, called the *Optimal Scheduling and Buffer Allocation*

(OSBA) problem. A similar linear programming formulation is possible for RSFGs as well. The key intuition to this idea is to obtain equations relating the variables $A[k, v]$'s and the buffer sizes required for the individual arcs. The equations are solved for $A[k, v]$'s subject to the condition that total buffer requirement for all the arcs is minimized. Further, in solving the linear equations, the optimal computation rate obtained using Reiter's results is used, ensuring rate-optimal schedules. The solution thus obtained guarantees optimal computation rate and minimal buffer requirement (if no register (buffer) sharing is allowed). We are currently working towards this goal.

The work reported in this paper considers only *regular* stream flow graphs. We plan investigate the scheduling of other stream flow graphs involving control actors such as switch and merge actors. Since the execution of these actors depend on the run-time value of the control tokens, a fully static schedule is not possible for these graphs. Quasi-static scheduling, proposed by Ha and Lee [14], is a possible direction to obtain optimal schedules for these graphs.

9 Acknowledgments

We wish to thank Prof. P. Panangaden, Marija Cubric and Qi Ning for the fruitful discussion we had with them. The authors are thankful to Erik R. Altman for his comments on the initial draft of this paper. The authors acknowledge Natural Science and Engineering Research Council (NSERC) and MICRONET, Network of Centers of Excellence for supporting this work. Finally the support received from the members of ACAPS (Advanced Computer Architecture and Program Structures Group) at McGill University is acknowledged.

References

[1] A. Aiken. *Compaction-Based Parallelization.* PhD thesis, Cornell U., 1988. Also publ. as Tech. Rep. TR 88-922, Dept. of Comp. Sci.

[2] A. Aiken and A. Nicolau. Optimal loop parallelization. In *Proc. of the SIGPLAN '88 Conf. on Programming Language Design and Implementation*, pages 308–317, Atlanta, Georgia, Jun. 1988.

[3] A. Aiken and A. Nicolau. A realistic resource-constrained software pipelining algorithm. In Alexandru Nicolau, David Gelernter, Thomas Gross, and David Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, chapter 14, pages 274–290. Pitman Pub. and the MIT Press, 1991. Selected papers from the Third Work. on Languages and Compilers for Parallel Computing, Irvine, Calif., Aug. 1990.

[4] G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proc. of the SIGPLAN '82 Symp. on Compiler Construction*, pages 98–105, Boston, Mass., Jun. 1982.

[5] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, Jan. 1981.

[6] L-F. Chao and E. H-M. Sha. Rate-optimal static scheduling for DSP dataflow programs. In *Proc. of 1993 Great Lakes Symp. on VLSI*, pages 80–84, Mar. 1993.

[7] J. B. Dennis. First version of a data-flow procedure language. In *Proc. of the Colloque sur la Programmation*, number 19 in Lec. Notes in Comp. Sci., pages 362–376. Springer-Verlag, 1975.

[8] K. Ebcioğlu. A compilation technique for software pipelining of loops with conditional jumps. In *Proc. of the 20th Ann. Work. on Microprogramming*, pages 69–79, Colorado Springs, Colorado, Dec. 1987.

[9] K. Ebcioğlu and A. Nicolau. A global resource-constrained parallelization technique. In *Conf. Proc., 1989 Intl. Conf. on Supercomputing*, pages 154–163, Crete, Greece, Jun. 1989.

[10] G. Gao, R. Govindarajan, and P. Panangaden. Well-behaved dataflow for DSP computation. In *Proc. of ICASSP-92, the 1992 Intl. Conf. on Acoustics, Speech, and Signal Processing*, volume I, pages 561–564, San Francisco, Calif., Mar. 1992.

[11] G. R. Gao, R. Govindarajan, and P. Panangaden. Construction rules for well-behaved stream programs. ACAPS Tech. Memo 26, Sch. of Comp. Sci., McGill U., Montréal, Qué., Apr. 1992.

[12] G. R. Gao, Q. Ning, and V. Van Dongen. Software pipelining for nested loops. ACAPS Tech. Memo 53, Sch. of Comp. Sci., McGill U., Montréal, Qué., May 1993.

[13] G. R. Gao, Y-B. Wong, and Q. Ning. A timed Petri-Net model for fine-grain loop scheduling. In *Proc. of the SIGPLAN '91 Conf. on Programming Language Design and Implementation*, pages 204–218, Toronto, Ontario, Jun. 1991.

[14] S. Ha and E. A. Lee. Compile-time scheduling and assignment of data-flow program graphs with data-dependent iteration. *IEEE Trans. on Computers*, 40(11):1225–1238, Nov. 1991.

[15] L. J. Hendren, G. R. Gao, E. R. Altman, and C. Mukerji. A register allocation framework based on hierarchical cyclic interval graphs. In U. Kastens and P. Pfahler, editors, *Proc. of the Intl. Conf. on Compiler Construction*, number 641 in Lec. Notes in Comp. Sci., pages 176–191. Springer-Verlag, Oct. 1992.

[16] P. N. Hilfinger. Silage reference manual, DRAFT release 2.0. Technical report, Dept. of Electrical Engineering and Comp. Sci., U. of Calif. at Berkeley, Jul. 1989.

[17] P. D. Hoang. *Compiling Real-Time Digital Signal Processing Applications onto Multiprocessor Systems*. PhD thesis, U. of Calif. at Berkeley, 1992. Publ. as Memorandum UCB/ERL M92/68.

[18] R. B. Jones and V. H. Allan. Software pipelining: An evaluation of enhanced pipelining. In *Proc. of the 24th Intl. Symp. and Work. on Microarchitecture*, pages 82–92, Albuquerque, N. Mex., Nov. 1991.

[19] R. M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics*, 23:309–311, 1978.

[20] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proc. of the SIGPLAN '88 Conf. on Programming Language Design and Implementation*, pages 318–328, Atlanta, Georgia, Jun. 1988.

[21] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Saunders College Publishing, Ft Worth, Tex., 1976.

[22] E. A. Lee, W.-H. Ho, E. Geoi, J. Bier, and S. S. Bhattacharyya. Gabriel: A design environment for DSP. *IEEE Trans. on Acoustics, Speech, and Signal Processing*, 37(11), Nov. 1989.

[23] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proc. of the IEEE*, 75(9):1235–1245, Sep. 1987.

[24] E. A. Lee. Consistency in dataflow graphs. *IEEE Trans. on Parallel and Distrib. Systems*, 2(2):223–235, Apr. 1991.

[25] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. on Computers*, 36(1):24–35, Jan. 1987.

[26] R. L. Lee. The effectiveness of caches and data prefetch buffers in large-scale shared memory multiprocessors. CSRD Rpt. No. 670, Center for Supercomputing Res. & Dev., U. of Ill. at Urbana-Champaign, May 1987.

[27] C. E. Leiserson, F. M. Rose, and J. B. Saxe. Optimizing synchronous circuitry by retiming. In *Proc. of the Third Caltech Conference on VLSI*, pages 87–116, Pasadena, Calif., Mar. 1983.

[28] L. E. Lucke and K. K. Parhi. Generalized ILP scheduling and allocation for high-level DSP synthesis. In *Proc. of the 1993 IEEE Custom Integrated Circuits Conf.*, 1993.

[29] A. Nicolau, K. Pingali, and A. Aiken. Fine-grain compilation for pipelined machines. Tech. Rep. TR 88-934, Dept. of Comp. Sci., Cornell U., Ithaca, N. Y., 1988.

[30] Q. Ning and G. Gao. Minimizing loop storage allocation for an argument-fetching dataflow architecture model. In *Proc. of PARLE '92 – Parallel Architectures and Languages Europe*, pages 585–600, Paris, France, Jun. 1992.

[31] Q. Ning and G. R. Gao. A novel framework of register allocation for software pipelining. In *Conf. Rec. of the Twentieth Ann. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 29–42, Charleston, South Carolina, Jan. 1993.

[32] Q. Ning and G. R. Gao. Optimal memory allocation for argument fetching dataflow machines. ACAPS Tech. Memo 32, Sch. of Comp. Sci., McGill U., Montréal, Qué., Jan. 1992.

[33] K. K. Parhi and D. G. Messerschmitt. Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding. *IEEE Trans. on Computers*, 40(2):178–195, Feb. 1991.

[34] D. B. Powell, W. C. Newmann, and E. A. Lee. Direct synthesis of optimized DSP assembly code from signal flow block diagrams. In *Proc. of ICASSP-92, the 1992 Intl. Conf. on Acoustics, Speech, and Signal Processing*, volume V, pages 553–556, San Francisco, Calif., Mar. 1992.

[35] H. Printz. *Automatic Mapping of Large Signal Processing Systems to a Parallel Machine.* PhD thesis, Carnegie-Mellon U., 1991. Publ. as Memorandum CMU-CS-91-101, Dept. of Comp. Sci.

[36] C. V. Ramamoorthy and G. S. Ho. Performance evaluation of asynchronous concurrent systems using Petri nets. *IEEE Trans. on Software Eng.*, 6(5):440–449, Sep. 1980.

[37] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proc. of the 14th Ann. Microprogramming Work.*, pages 183–198, Chatham, Mass., Oct. 1981.

[38] R. Reiter. Scheduling parallel computations. *J. of the ACM*, 15(4):590–599, Oct. 1968.

[39] S. Ritz, M. Pankert, and H. Meyr. High level software synthesis for signal processing systems. In *Proc. of the 1992 Intl. Conf. on Application Specific Array Processors*, Berkeley, Calif., Aug. 1992. IEEE Comp. Soc.

[40] G. C. Sih. *Multiprocessor Scheduling to Account for Interprocessor Communication.* PhD thesis, U. of Calif. at Berkeley, 1991. Publ. as Memorandum UCB/ERL M91/29.

[41] L. Thiele. Resource constrained scheduling of uniform algorithms. In *Proc. of the 1993 Intl. Conf. on Application Specific Array Processors*, pages 29–40, Venice, Italy, Oct. 1993. IEEE Comp. Soc.

[42] R. F. Touzeau. A Fortran compiler for the FPS-164 scientific computer. In *Proc. of the SIGPLAN '84 Symp. on Compiler Construction*, pages 48–57, Montréal, Qué., Jun. 1984.

[43] V. Van Dongen, G. R. Gao, and Q. Ning. A polynomial time method for optimal software pipelining. In *Proc. of the Conf. on Vector and Parallel Processing, CONPAR-92*, number 634 in Lec. Notes in Comp. Sci., pages 613–624, Lyon, France, Sep. 1–4, 1992. Springer-Verlag.

[44] M. Čubrić and P. Panangaden. Minimal memory schedules for dataflow networks. In Eike Best, editor, *Proc. of the 4th Intl. Conf. on Concurrency Theory*, number 715 in Lec. Notes in Comp. Sci., pages 368–383, Hildesheim, Germany, Aug. 23–26, 1993. Springer-Verlag.

A Transformation of RSFGs into Homogeneous Dataflow Graphs

This section deals with the transformation of RSFGs into equivalent homogeneous dataflow graphs. The crux of the above issue is a careful definition of "equivalent". In [10], equivalence of an RSFG and a homogeneous dataflow graph is defined as:

Definition A.1 *Let R be a regular actor with I_R and O_R as its input and output channels. Suppose that when R fires once, it consumes $\vec{N}$ (an I_R indexed vector of positive integers) and produces $\vec{M}$ (an O_R indexed vector of positive integers) tokens. Let H be a homogeneous actor with I_H as its input channels and O_H as its output channels. We say that H* **emulates** *the firings of R if there exists mappings $\mathcal{M}_I : I_H \rightarrow I_R$ and $\mathcal{M}_O : O_H \rightarrow O_R$ such that for each $i \in I_R$, the number of distinct channels of H mapped to i by $\mathcal{M}_I$ is $\vec{N}[i]$ and similarly for $\mathcal{M}_O$.*

The intuition here is that each input channel i of the regular actor R is emulated by i_k input channels of the homogeneous actor, where i_k is the number of tokens consumed on i. A similar thing holds for the output channels. The input mapping $\mathcal{M}_I$ ensures that the number of input tokens required for one firing of H is sufficient to satisfy the input requirements for the regular actor. Likewise, $\mathcal{M}_O$ guarantees the necessary output tokens are produced on the output channels of R.

From the above definition of "equivalent", it can be argued that the optimal computation rate for the homogeneous graph is the optimal rate of a *balanced* schedule for the RSFG. Further, it has been shown in [10] it is always possible to transform a well-behaved RSFG into an equivalent homogeneous graph. The following theorem makes precise the conditions under which the claim is true.

Theorem A.1 *An RSFG can be transformed into an equivalent homogeneous dataflow graph if and only if the state equations of the RSFG have a non-trivial solution.*

The transformation procedure is informally described in [40] for synchoronous dataflow graphs which are equivalent to RSFGs. Algorithm A.1 formalizes the construction of equivalent homogeneous graphs.

Algorithm A.1

[Input] A regular graph $G = (V, E, m, d)$, and the solution vector $[k_{v_1},\ k_{v_2},\ \ldots,\ k_{v_n}]$ for the state equations of G
[Output] An equivalent homogeneous graph $\hat{G} = (\hat{V}, \hat{E}, \hat{M}, \hat{D})$

Step 1 : Initialize $\hat{V} = \phi$, $\hat{E} = \phi$, where ϕ represents the empty set.

Step 2 : For each $v_i \in V$ do

Step 2.1 : Add nodes $v_i^0,\ \ldots,\ v_i^{(k_{v_i}-1)}$ to the set $\hat{V}$.
Step 2.2 : For each $k \in [0, k_{v_i})$ do
Step 2.2.1 : Set $k' = (k+1)\ \ mod\ \ k_{v_i}$
Step 2.2.2 : Include an edge $(v_i^k, v_i^{k'})$ in the set $\hat{E}$.
Step 2.2.3 : Set $\hat{M}_{(v_i^k, v_i^{k'})} = \lfloor \frac{(k+1)}{k_{v_i}} \rfloor$.
Step 2.2.4 : Set $\hat{D}_{v_i^k} = d_{v_i}$

Step 3 : For each $(v_j, v_i) \in E$ do

Step 3.1 : Set $k' = \lfloor \frac{(m_{v_j v_i}\ \ mod\ \ (k_{v_i} * I_{v_j v_i})}{I_{v_j v_i}} \rfloor$;
Step 3.2 : Set $k'' = \Big(m_{v_j v_i}\ \ mod\ \ (k_{v_i} * I_{v_j v_i})\Big)\ \ mod\ \ I_{v_j v_i}$;
Step 3.3 : For $k = 0$ step 1 to $k_{v_j} - 1$ do
Step 3.3.1 For each $l = 1$ step 1 to $O_{v_j v_i}$ do
Step 3.3.1.1 : Include an edge $(v_j^k, v_i^{k'})$ in the set $\hat{E}$.
Step 3.3.1.2 : Set $\hat{M}_{v_j^k, v_i^{k'}} = \lceil \frac{m_{v_j v_i}}{k_{v_i} * I_{v_j v_i}} \rceil$
Step 3.3.1.3 : Set $m_{v_j v_i} = max\Big((m_{v_j v_i} - 1), 0\Big)$
Step 3.3.1.4 : Set $k' = \Big(k' + \lfloor \frac{k''+1}{I_{v_j v_i}} \rfloor\Big)\ \ mod\ \ k_{v_i}$
Step 3.3.1.5 : Set $k'' = (k''+1)\ \ mod\ \ I_{v_j v_i}$

We will calculate the number of nodes and the number of edges in the equivalent homogeneous graph. It can be seen (from Step 2.1) that the homogeneous graph for G consists of $\sum_{v_i \in V} k_{v_i}$ nodes. Step 2.2.2 adds k_{v_i} edges to the homogeneous graph for each v_i. Thus there are $\sum_{v_i \in V} k_{v_i}$ such arcs, referred to as the self dependency arcs. Further, Step 3.3.1.1 contributes one edge to $\hat{E}$ and this step is iterated $k_{v_i} * I_{v_j v_i}$ times for each edge (v_j, v_i) in the original RSFG. Thus, there is a total of $\sum_{(v_j, v_i) \in E} k_{v_i} * I_{v_j v_i}$ such edges in the homogeneous graph. Therefore the total number of edges in the homogeneous graph is given by

$$\sum_{v_i \in V} k_{v_i} + \sum_{(v_j, v_i) \in E} k_{v_i} * I_{v_j v_i}.$$

Time	iteration 0	iteration 1	iteration 2
0	a		
1	a		
2	b		
3	a,c		
4	b		
5	c		
6	c		
7		a	
8		a	
9		b	
10		a,c	
11		b	
12		c	
13		c	
14			a
15			a
16			b
17			a,c
18			b
19			c
20			c

Figure 10: A Non-Overlapped Schedule for the RSFG in Example 3.1

B Block Scheduling of RSFGs

In this section we present the optimal schedules for the graphs used in Example 3.1 and 3.2.

B.1 Non-Overlapped Schedule for Example 3.1

The optimal non-overlapped schedule for the RSFG of Example 3.1 is shown in Fig. 10. As can be observed from the figure, the computation rate of this schedule is $\frac{1}{7}$.

By performing retiming [27], it is possible to construct the following non-overlapped schedule which is rate-optimal[10].

B.2 Non-Overlapped Schedule for Example 3.2

The block scheduling approach [26] does not perform any unfolding transformation. The optimal non-overlapped schedule for the RSFG of Example 3.2 is shown in Fig. 12. The computation rate of this schedule is $\frac{1}{5}r$, or the iteration period is 5 time units.

[10] A description of the retiming operation and its relevant properties are beyond the scope of this paper and the reader is referred to [27].

Time	iteration 0	iteration 1	iteration 2
0	a,c		
1	a,c		
2	b,c		
3	a		
4	b		
5		a,c	
6		a,c	
7		b,c	
8		a	
9		b	
10			a,c
11			a,c
12			b,c
13			a
14			b

Figure 11: A Non-Overlapped Rate-Optimal Schedule for Example 3.1

Time	iteration 0	iteration 1	iteration 2
0	a		
1	a		
2	b		
3	c		
4	&		
5		a	
6		a	
7		b	
8		c	
9		&	
10			a
11			a
12			b
13			c
14			&

Figure 12: A Non-Overlapped Schedule for the RSFG of Example 3.2

Time	iteration 0	iteration 1	iteration 2
0	a,b		
1	a,c		
2	&		
3		a,b	
4		a,c	
5		&	
6			a,b
7			a,c
8			&

Figure 13: A Non-Overlapped Schedule for the RSFG after Retiming

By retiming the RSFG, it is possible to obtain the following schedule which is better, in terms of computation rate, than the schedule shown in Fig. 12. The computation rate of the schedule of Fig. 13 is $\frac{1}{3}$ and therefore is not rate-optimal.