

Buffer Allocation in Regular Dataflow Networks: An Approach Based on Coloring Circular-Arc Graphs

R. Govindarajan

Supercomputer Education and Research Center
Dept. of Computer Science & Automation
Indian Institute of Science
Bangalore, 560 012, India
govind@serc.iisc.ernet.in

S. Rengarajan

BFL Software Limited
Gopalakrishna Complex
45/3 Residency Road
Bangalore, 560 025, India
rengs@peritus.com

Abstract

In this paper we discuss a method to perform compile-time buffer allocation, allowing efficient buffer sharing among the arcs of a special form of dataflow graphs – known as regular stream flow graphs – commonly used in Digital Signal Processing applications. We relate the buffer sharing problem to that of finding *independent sets* in **weighted circular arc graph**. An important difference between the traditional graph coloring/register allocation problem and our buffer sharing problem is that in our problem the aim is to minimize the sum of the weights of the independent sets, rather than the number of independent sets. We present a heuristic algorithm and experiment it on a large number of randomly generated regular dataflow graphs as well as a few DSP applications. It is observed that the heuristic algorithm performs well, reducing the buffer requirement by 14.3% on the average. Also, we observe that buffer requirement achieved by the heuristic algorithm is within 2.1% from the lower bound.

1 Introduction

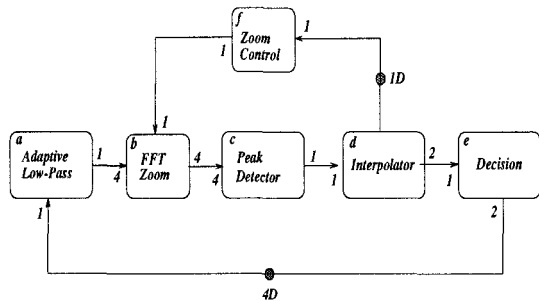
In this paper we consider a class of dataflow graphs, called synchronous dataflow graphs [12, 15] or Regular Stream Flow Graphs (RSFGs) [4], which have been widely accepted as a powerful programming model for Digital Signal Processing (DSP) applications. Nodes of these dataflow graphs represent large-grain tasks and produce (or consume) multiple, sometimes in tens or even in hundreds, of data values per invocation. Each of these tasks represent operations such as an FIR filter, FFT, or bandshifting. The number of data values produced (consumed) by each task is known a priori and is fixed. The term “tokens” is also used to

refer to the data values produced by the nodes. The numbers of data tokens produced by the nodes of the regular dataflow graph are different and hence the firing rates of the nodes may be different. Hence these graphs are also known as *multi-rate* graphs.

A number of methods have been proposed to obtain compile time schedules for RSFGs or synchronous dataflow graphs [3, 6, 7, 8, 12, 14, 15]. In this paper, we discuss a method to perform buffer allocation for the compiled schedules of the RSFG. The method is uniformly applicable to any of the scheduling methods mentioned earlier. In allocating buffers for the arcs of the RSFG we identify arcs, to be more precise *live ranges* of arcs, which can share the same buffer. Two arcs can share the same buffer (i.e. use the same memory space) if their live ranges do not overlap. This problem is similar to the traditional register allocation problem [1, 2]. In traditional register allocation each live range requires a single register. However, in our RSFGs, since multiple data values can be produced by each firing of a node, the buffer (or memory) required for an arc is greater than unity. This gives rise to a significantly different problem which, to the best of our knowledge, has not yet been studied.

Like the traditional register allocation problem, buffer allocation problem can also be reduced to the problem of graph coloring. However, in our case, (i) the graph is a *circular-arc graph* [5], and (ii) the nodes of the circular-arc graph have an associated weight equal to the buffer size required for the arc. Hence we call these graphs, *weighted circular-arc graphs*. Our aim is to find the sets of nodes (of the graph) which can share the same buffer; that is, find the *independent sets* [5]¹. Unlike the traditional graph coloring problem where the objective is to minimize the number of

¹See Section 3.1 for a definition of independent sets, and weights of the independent sets.



(a) RSFG for Spectrum Analyzer

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

(b) A Rate-Optimal Schedule

Figure 1: A Motivating Example

independent subsets, our aim is to minimize the sum of the weights of the independent subsets.

We propose a simple heuristic algorithm which computes the independent sets. We test our heuristic buffer sharing algorithm on a large number (more than 1000) of randomly generated RSFGs and their compiled schedules. The buffer sharing algorithm reduces the buffer requirement by 14.3% on the average. The median value for the improvement in memory requirement is 12.97%, while the maximum improvement achieved for some schedule is as high as 57.4%. Lastly we establish a lower bound for the buffer requirement under buffer sharing. We formally establish that this is a loose lower bound. Experimental results show that the buffer requirement achieved by our algorithm deviates from the lower bound by less than 2.1% on the average (arithmetic mean). The median value for this is only 0.7%. Further, in 38.5% of the test cases, our algorithm achieves the lower bound, i.e. it achieves the optimal buffer allocation.

In the following section we present a motivating example. In Section 3, we model the buffer allocation problem using *weighted circular-arc graphs* and present a heuristic algorithm for buffer sharing. Experimental results are reported in Section 4. Related work and conclusions are presented in Sections 5 and 6.

2 Background and Motivation

In this section we motivate the buffer allocation problem with the help of an example.

2.1 Repetitive Schedules for RSFGs

Throughout this paper we assume that a computation is represented by a Regular Stream Flow Graph (RSFG) [4]. As an example, Fig. 1(a) shows the RSFG for a spectrum analyzer. Node *c* which corresponds to

a Peak Detector, for example, consumes 4 data values and produces 1 data value for each firing². The nodes of this RSFG fire at different rates; i.e. nodes *a*, *b*, *c*, *d*, *e*, and *f* fire, respectively 4, 1, 1, 1, 2, and 1 times. Informally, the above firings of nodes of the RSFG constitute an *iteration*. Feedback arcs, e.g. arc (*d*, *f*), may carry initial data samples, marked by a dot on the arcs.

Let the execution time of actor *b* be 2 time units while that of other actors be unity. A schedule for the RSFG is shown in Fig. 1(b) where the nodes of the RSFG in a same column fire concurrently. The symbol *_b* is used in Fig. 1(b) to indicate that the execution of actor *b* is continuing from the previous time step. A repetitive pattern, known as the software-pipelined schedule can be observed from time step 2 to time step 9. The length of the repetitive pattern is the *period* of the schedule. Lastly, the schedule shown in Fig. 1(b) is *rate-optimal*, since this is the fastest execution rate of the above RSFG, given the above dependence constraints.

2.2 Buffer Allocation

In this paper we consider only the buffer requirements of the repetitive pattern of the schedule. Since the prologue is executed only once, separate buffer allocation can be done for it. In computing the buffer requirements for the arcs of the RSFG, we make the following assumption without any loss of generality. Input tokens remain on the input arc until the activation (firing) is completed and output tokens are produced (all at once) at the end of the firing.

²For brevity, we have scaled the number of tokens produced/consumed by a node in the spectrum analyzer example. However, tasks such as Peak Detector, will consume as many as 256 input samples to produce a single output. In our experiments we used the actual sample size without any scaling.

Next we compute the buffer requirement for each arc of the RSFG using an operation model. This is done by simulating the schedule shown in Fig. 1(b) and computing the maximum number of tokens on each of the arcs during the repetitive pattern. One can verify that the buffer requirements of the arcs of the RSFG are as shown in the following table.

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

2.3 Buffer Sharing

In this section we present an approach to reduce the memory requirement of the RSFGs by allowing buffer sharing. We begin by first defining the live range of an arc. The live range of an arc starts at a point of time when the number of tokens in the associated buffer becomes greater than zero. The live range ends when the number of tokens in the associated buffer becomes zero again. The live range of each arc of the RSFG and the sizes of the associated buffers during the various time steps in the repetitive pattern are shown in Fig 2. If the buffer associated with an arc contains at least one token throughout the repetitive pattern, we say that the live range of the arc spans the repetitive pattern. If the live range of an arc goes across iterations, we represent the live range by means of a circular arc [9]. As an example consider the live range of arc (d, e) which extends from time step 8 to 1. Finally, in an RSFG, it is possible that an arc may have multiple disjoint live ranges. In this paper, we associate a buffer to each live range rather than to each arc.

From Fig. 2, one can observe that the live ranges of arc (a, b) do not overlap with that of (b, c). Hence, instead of allocating individual buffer space for these two arcs, they can be allowed to share the same memory space. Further, the same buffer can also be shared by the live ranges of arc (c, d) and (d, e). Under buffer sharing, the total buffer size allocated for these live ranges is the maximum of the buffer size required for each of them, in this case 4. As another example, the live ranges of (d, f) and (f, b) do not overlap, and hence they can share the same buffer. However, the live range of (e, a) cannot share its buffer with (a, b).

One can identify, for the given schedule (shown in Fig. 1(b)), the following buffer sharing is possible: Arcs (a, b), (b, c), (c, d), and (d, e) can share the same buffer space and requires a size of 4; arcs (d, f) and (f, b) can share the same buffer space and requires a size of 1; lastly arc (e, a) requires a buffer size of 3. Thus, the total memory requirement for the given schedule,

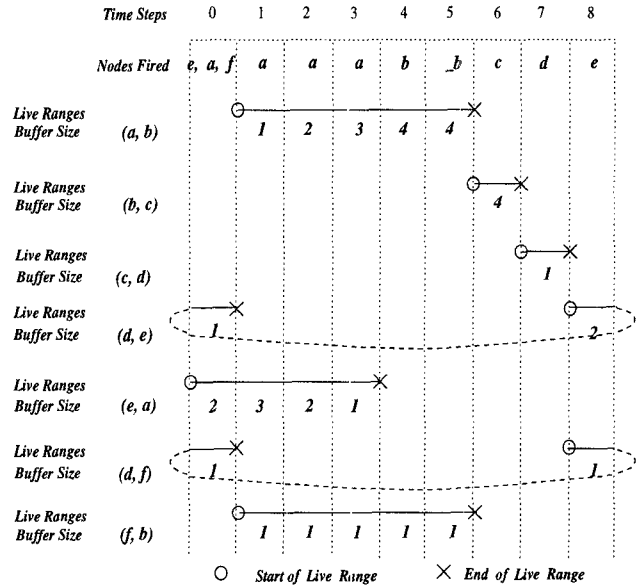


Figure 2: Live Ranges of Example Arcs

under buffer sharing, is only 8. Thus buffer sharing significantly reduces the buffer requirement from 16 to 8.

3 Buffer Sharing using Circular Arc Graph Coloring

In the following subsection First, we relate the buffer sharing problem to finding the independent sets in a (weighted) circular-arc graph. In Section 3.2 we present a heuristic algorithm for the buffer sharing problem. A lower bound of the buffer requirement is established in Section 3.3.

3.1 Circular Arc Graphs

The buffer sharing problem can be stated as:

Given an RSFG and a schedule for it, derive a buffer allocation, allowing buffer sharing wherever possible, such that the sum of buffer sizes allocated for all the live ranges is minimized.

From the live ranges of the arcs of the RSFG, we derive an interference graph $G = (V, E, w)$. A vertex $v \in V$ represents a live range, and an edge $e \in E$ between two vertices v and v' indicates that the respective live ranges overlap. The function w associates a weight $w(v)$, equal to the size of the buffer required for the corresponding live range, with vertex v . The interference graph for our motivating example is shown

in Fig. 3. Vertices v_1 to v_7 correspond to the live ranges of arcs (a, b) , (b, c) , (c, d) , (d, e) , (e, a) , (d, f) , and (f, b) respectively. The number inside a node represents the weight of that node. The interference graph referred to here conforms to the definition of an *interval graph* [5], or more precisely, with the presence of circular live ranges, a *circular-arc graph*. Henceforth, we use the term ‘circular-arc graph’ and ‘interference graph’ interchangeably. Likewise, the terms ‘interval’ and ‘live ranges’ are also used interchangeably.

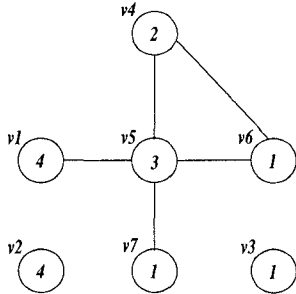


Figure 3: Interval Graph for the Motivating Example

Next we define the following terms [5].

Definition 3.1 A subset S of vertices is an **independent set**, if there exists no edge between any two vertices in S .

Definition 3.2 An independent set is said to be **maximal** if it is not contained in any other independent set.

In our circular-arc graph, $\{v_2, v_5\}$ is an independent set while $\{v_1, v_2, v_3, v_4, v_7\}$ is a maximal independent set.

Now, the buffer sharing problem can be recognized as partitioning the set of vertices V into independent sets V_1, V_2, \dots, V_m . Since there does not exist an edge between any pair of vertices in an independent set V_i , the vertices in V_i can share their buffers. We define the weight of an independent set to be the maximum of the weights of each vertex in V_i . That is

$$w(V_i) = \max_{v \in V_i} (w(v))$$

Finally, the objective of the buffer sharing problem is to minimize the total buffer requirement. This is same as minimizing the sum of the weights of the independent sets.

3.2 A Heuristic Algorithm for Buffer Sharing

Our algorithm proceeds by first sorting the vertices of the interference graph in the descending order based

on their weights. Starting with the first vertex v (with the maximum weight) in the sorted list we determine the maximal independent set. In determining the maximal independent set, we try to include the nodes with the largest weight. That is, we traverse every other vertex, in the sorted order, to check whether it can be included in the independent set. Once a maximal independent set is found, we remove these vertices from V . We proceed to find the next maximal independent set in the remaining vertices in a similar way. We continue in this manner, until all vertices in the circular-arc graph are included in one of the independent sets.

The algorithm is formally described below.

Algorithm 3.1

Input: An interference graph $G = (V, E, w)$.

Output: Independent sets V_1, V_2, \dots, V_m with the objective to minimize the sum of the weights of the subsets.

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

We observe that Algorithm 3.1 is greedy and tries to share buffers for vertices with maximum weight first. This may not always result in an optimal solution.

3.3 A Lower Bound for Buffer Requirement

In this section we establish a lower bound for the buffer requirement, of an RSFG under buffer sharing. It is easy to see that whenever two live ranges v_1 and v_2 overlap, two distinct buffers b_1 and b_2 are needed. The notion of *cliques* in an undirected graph is helpful in establishing a simple lower bound [5].

Application	RSFG size		Period of the Schedule	Buffer Requirement			Improvement due to Sharing	Deviation from Lower Bound
	# nodes	# arcs		Without Sharing	With Sharing	Lower Bound		
Phase-Locked Loop	10	12	5	23	22	22	4.35	0.0
Voice-Band	7	14	16	30	22	22	26.67	0.0
Power Spectrum	9	10	128	180	164	164	8.89	0.0
Auto Correlation	16	17	256	364	292	292	19.78	0.0
Periodogram	7	7	256	245	245	245	0.00	0.0
Comb Filter	21	26	2	32	24	24	25.00	0.0
Satellite Receiver	19	25	151	625	523	309	13.32	69.3
Spectrum Analyzer	6	7	84	52	34	34	34.62	0.0

Table 1: Performance of our Buffer Sharing Algorithm on DSP Applications

Definition 3.3 A subset S of vertices of an undirected graph G is a **clique**, if there exists an edge between every pair of vertices in S .

Definition 3.4 A clique is said to be a **maximal clique** if it is not contained in any other clique.

Examples of maximal cliques in Fig. 3 are $\{v1, v5\}$, $\{v5, v4, v6\}$, and $\{v5, v7\}$. Thus, nodes of a clique cannot share their buffer space among themselves. Thus the total buffer requirements for the nodes of a clique is equal to the sum of the buffer requirements of the individual nodes in that clique. Hence, the clique that has the maximum buffer requirement governs the lower bound. The following lemma formalizes this.

Lemma 3.1 The minimum buffer requirement is at least as large as the maximum of the weights of the cliques in the interference graph.

In our motivating example, the clique $\{v1, v5\}$ has the maximum weight $4 + 3 = 7$. We remark that this lower bound is not tight. It can be seen that the maximal independent sets $\{v1, v2, v3, v4\}$, $\{v6, v7\}$, $\{v5\}$ result in the minimum buffer requirement of 8 for the RSFG, while the lower bound is only 7. Lastly, we reemphasize that our objective is minimizing the sum of the weights of the independent sets and not minimizing the number of independent sets. For example we are not interested in the partition, $\{v1, v2, v3, v6\}$, $\{v4, v7\}$, $\{v5\}$ having only 3 independent sets, because the sum of the weight of the independent sets is $4 + 2 + 3 = 9$.

4 Experimental Results

In this section we present some experimental results of our buffer sharing algorithm.

In our experimental setup we generate random RSFGs and determine rate-optimal schedules for them as

discussed in [6]. We supplement this study with results obtained from a set of DSP algorithms.

We ran the experiments on more than 1000 randomly generated RSFGs. The buffer sharing algorithm reduced the memory requirement of the schedules in 94.7% (or 947 out of 1000) test cases by 14.3% on the average (arithmetic mean). The median value for the improvement in buffer requirement achieved by our buffer sharing algorithm is 12.97% while the maximum improvement achieved for some schedule is as much as 57.4%. In 7 of the 8 DSP applications a reduction in buffer requirement by 16.95% was achieved by our buffer sharing algorithm. Table 1 gives the performance of our buffer sharing algorithm for the various DSP applications.

The buffer sharing algorithm achieves the lower bound, i.e. the optimal solution, in 384 out of 1000 test cases (38.4%). On the average, the heuristic algorithm is within 2.1% from the lower bound. The median for this value is 0.66%. At worst, the heuristic algorithm is 26% from the lower bound; however in more than 85% of the test cases it is only with 5% and in 95% of the test cases it is within 10% from the lower bound.

5 Related Work

Earlier work on regular dataflow networks concentrated on obtaining efficient schedules. Several other scheduling methods, some of them are based on the block scheduling method with different objectives, have been proposed by Lee's research group (see e.g. [12, 8]). In [3], algorithms to construct either buffer optimal or maximally concurrent schedules are reported. Methods to construct rate-optimal schedules have been proposed in [6, 7, 14, 10].

The work proposed in this paper complements the work reported in earlier literature in the sense that it attempts to minimize the buffer requirements of the

constructed schedule by identifying live ranges which can share their buffers. It is important to note that the buffer optimal schedules proposed in [3, 7] do not consider buffer sharing. Hence our buffer sharing algorithm can be applied even to the buffer optimal schedules obtained from these methods [3, 7] and significant reduction in memory requirement can be achieved.

The buffer sharing algorithm considered in this paper closely resembles graph coloring or identifying maximal independent sets in interval graphs [5]. More recently, a number of efficient algorithms for finding maximal independent sets for subclasses of *perfect graphs* have been proposed [11, 13]. Chaitin, et. al., have used interval graph coloring method to solve the register allocation problem in compilers [2]. However, all the above work concentrate on unweighted graphs.

6 Conclusions

In this paper we have proposed a method for reducing the buffer requirements of a regular stream flow graph (RSFG). Our approach is to allow sharing of buffers among the arcs whose live ranges do not overlap. Traditionally, this problem has been related to coloring interval graphs. In our case, as sizes of the individual buffers may be greater than unity, the problem reduces to coloring a weighted circular-arc graph. To the best of our knowledge, though there exists many solutions for the traditional graph coloring problem, there does not exist any solution when the nodes of the graphs have associated weights.

We have proposed a heuristic algorithm to perform buffer sharing. We have also established a simple lower bound for the memory requirement. The heuristic algorithm was implemented and tested on 1000 randomly generated RSFGs as well as 8 DSP applications. In 95% of the test cases, the buffer sharing reduces the buffer requirement by 14.3% on the average. Our algorithm achieved the lower bound in 38.47% of the test cases. Finally the heuristic algorithm was within 2.1% on the average from the lower bound.

Acknowledgments

We thank Prof. Guang R. Gao, Palash Desai, and Dr. S. Ritz for their comments and help.

References

- [1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers — Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, MA, 1988.
- [2] 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.
- [3] M. Čubrić and P. Panangaden. Minimal memory schedules for dataflow networks. In *Proc. of the 4th Intl. Conf. on Concurrency Theory*, LNCS:715, pp. 368–383, Hildesheim, Germany, Aug. 1993.
- [4] G. R. Gao, R. Govindarajan, and P. Panangaden. Well-behaved dataflow programs for DSP computation. In *Proc. of the 1992 Intl. Conf. on Acoustics, Speech, and Signal Processing*, volume V, pp. 561–564, San Francisco, CA, Mar. 1992.
- [5] M.C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, 1980.
- [6] R. Govindarajan and G. R. Gao. A novel framework for multi-rate scheduling in DSP applications. *Proc. of the 1993 Intl. Conf. on Application Specific Array Processors*, pp. 77–88. Venice, Italy, Oct. 1993.
- [7] R. Govindarajan, G. R. Gao, and P. Desai. Minimizing memory requirements in rate-optimal schedules. *Proc. of the 1994 Intl. Conf. on Application Specific Array Processors*, pp. 75–86, San Francisco, CA, Aug. 1994.
- [8] 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.
- [9] L. J. Hendren, G. R. Gao, E. R. Altman, and C. Mukerji. A register allocation framework based on hierarchical cyclic interval graphs. *Proc. of the 4th Intl. Conf. on Compiler Construction, CC '92*, LNCS:641, pp. 176–191, Paderborn, Germany, Oct. 1992.
- [10] L-G. Jeng and L-G. Chen. Rate-optimal DSP synthesis by pipeline and minimum unfolding. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 2(1):81–88, Mar. 1994.
- [11] T. Kashiwabara, S. Masuda, K. Nakajima, and T. Fujisawa. Generation of maximum independent sets of a bipartite graph and maximum cliques of a circular arc graph. *Jl. of Algorithms*, 13:161–174, 1992.
- [12] 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.
- [13] J.Y.T. Leung. Fast algorithms for generating all maximum independent sets of interval, circular-arc and chordal graphs. *Jl. of Algorithms*, 5:22–35, 1984.
- [14] 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.
- [15] 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, CA, Aug. 1992.