Scheduling Optimisations for SPIN to Minimise Buffer Requirements in Synchronous Data Flow

Pieter H. Hartel and Theo C. Ruys
University of Twente, The Netherlands
email: {P.H.Hartel,T.C.Ruys}@utwente.nl

Marc C. W. Geilen
Eindhoven University of Technology, The Netherlands
email: M.C.W.Geilen@tue.nl

Abstract—**Synchronous Data flow (SDF) graphs have a simple and elegant semantics (essentially linear algebra) which makes SDF graphs eminently suitable as a vehicle for studying scheduling optimisations. We extend related work on using SPIN to experiment with scheduling optimisations aimed at minimising buffer requirements. We show that for a benchmark of commonly used case studies the performance of our SPIN based scheduler is comparable to that of state of the art research tools. The key to success is using the semantics of SDF to prove when using (even unsound and/or incomplete) optimisations are justified. The main benefit of our approach lies in gaining deep insight in the optimisations at relatively low cost.**

I. INTRODUCTION

Synchronous Data Flow (SDF) is a paradigm for describing Digital Signal Processing (DSP) applications [13]. SDF has a long history dating back to the early 70s. Mainly due to the ever increasing interest in embedded systems, SDF is currently an active area of research. A typical application processes an infinite stream of data samples, which enter the SDF graph at the source node(s), and which exit the graph at the sink node(s). The SDF formalism abstracts away from the actual calculations taking place at the nodes, the contents of the tokens, and the time taken to transfer tokens or to perform calculations.

An SDF graph is a directed, connected graph. Each node in the graph represents a processing step, and the edges transport tokens between nodes. The nodes may fire independently of each other, and concurrently. The term synchronous means that when a node fires, it always consumes the same number of tokens from each input port, and the node always produces the same number of tokens on each output port. Each edge is connected to precisely one producer and precisely one consumer. A node that does not consume tokens is a source node, and a node that does not produce tokens is a sink node. An SDF graph may be cyclic. An SDF graph cannot be used to represent conditionals (this would make the SDF asynchronous). The semantics of an SDF graph can be given using linear algebra.

SDF graphs come in many flavours; we focus on the classical variant as discussed by Lee and Messerschmitt [13].

a) Problem: There are special purpose analysis tools that optimise throughput, latency, buffer requirements, timing and other relevant architectural parameters of an SDF graph as part of the DSP design flow. Even though the optimisation
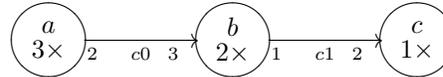

Fig. 1. Simple SDF graph.

problems are typically NP complete [14], the simple semantics of SDF makes it possible to prove a wealth of useful properties that can be used as optimisations in the analysis. However, designing the algorithms, and experimenting with the optimisations requires a significant amount of effort.

b) Contribution: We show that due to the semantic simplicity of the SDF graph it is feasible to use a model checker as an efficient analysis tool for buffer requirements, making it easy to experiment with various optimisations. Such experiments are more difficult to conduct with a special purpose tool than with a powerful general purpose tool. The optimisations themselves are not specific to the model checker but can be applied in any other setting. We build on work from Geilen, Basten and Stuijk [7] (henceforth referred to as GBS) focusing on minimising the buffer space required for the channels. We improve the work of GBS in two ways. Firstly, we explore improvements to the efficiency of *checking* the minimum bounds, both in case the channel buffers share a common area of memory and in the case where each channel buffer has a separate area of memory (see Sections III . . . VI). Secondly, we develop new theory and the algorithms necessary for *finding* the minimum bounds (Section VII) for the common buffer case.

II. EXAMPLES

To give the intuition for the semantics of SDF we discuss three examples, the first of which is shown in Figure 1. The SDF graph has three nodes a, b, and c and two edges $c0$ and $c1$. The number at the tail of an edge is the production rate; the number at the head of an edge is the consumption rate. Node a is the source, and node c is the sink. The number in the node (e.g. $3\times$) is the relevant component of the repetition vector as calculated by Equation 4. Figure 1 is actually a chain, which is a directed connected graph of k nodes and $k-1$ edges such that only one path exists from the first to the last node [1, Chapter 4].

Each time node a fires, two tokens are produced and sent on channel $c0$ to node b. Node a must fire at least twice before

Fig. 2. Cyclic SDF graph (left) and an inconsistent SDF graph (right).

node b is able to fire, because b consumes 3 tokens. Similarly, b must fire at least twice before c is able to fire. The state of the system records the current number of tokens on each channel. Firing a node causes the system to make a state transition. A periodic schedule is a sequence of state transitions that, starting from an initial state brings the system back into the initial state. The SDF graph of Figure 1 admits infinitely many periodic schedules. The shortest periodic schedules for our example are $(aababc)^*$ and $(aaabbc)^*$. These schedules are actually sequential schedules. In the first schedule the data dependencies inhibit concurrency, in the second schedule a and b may fire concurrently: $(aa(a||b)bc)^*$. Following GBS, in the sequel we will focus on sequential schedules. (Parallel schedules never require less buffer space than sequential schedules.) The minimum buffer capacity for $c0$ required by the second (sequential) schedule is 6 tokens, whereas for the first schedule 4 tokens would suffice on $c0$. Therefore schedule $(aababc)^*$ is the best of the two schedules in terms of the buffer capacity for $c0$.

The second example (Figure 2 left) shows a cyclic graph with two nodes d and e. Unlike the previous example, in which data can flow directly, this example is deadlocked, unless some initial tokens are present. Assume that 2 initial tokens are present on $c2$, as indicated by the two bullets. Then node e can fire twice, producing a total of 4 tokens on $c3$, after which node d can fire, once. This brings the system back in the initial state. This time the only possible schedule is: $(eed)^*$. The minimum buffer capacity required for $c2$ is 2 and 4 for $c3$.

The third example (Figure 2 right) shows an inconsistent SDF graph. The problem is that each time node f fires, it places 2 tokens on $c4$ and only one token on $c5$, whereas node g removes one token from both channels. This means that tokens will continue to accumulate on $c4$, which thus requires an infinite buffer capacity for any periodic (hence non-terminating) schedule; this is infeasible.

III. SEMANTICS

An SDF graph with N nodes and C channels can be characterised by a *topology matrix*, with C rows and N columns, where the entries of the matrix give the production rates (positive) and consumption rates (negative) of the SDF graph. The topology matrix Γ for Figure 1 is:

$$\Gamma = \begin{bmatrix} 2 & -3 & 0 \\ 0 & 1 & -2 \end{bmatrix}$$

The state vector $\vec{s}(i)$ of the system is a non-negative column vector (of height C) representing the number of tokens held in each channel after i nodes have fired. The initial state $\vec{s}(0)$

specifies the number of tokens initially present on the channels, for example:

$$\vec{s}(0) = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{1}$$

A state transition consists of two steps. Firstly a non-deterministic choice is made to select the node that is to be fired. This choice is represented in the column vector $\vec{f}(i)$:

$$\vec{f}(i) = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \text{ or } \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \text{ or } \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{2}$$

Secondly, the effect of firing the node on the state is specified by Equation 3, making sure that firing the selected node maintains a non-negative state vector (we ignore self edges, as the required buffer size for a self edge is easy to calculate):

$$\vec{s}(i+1) = \vec{s}(i) + \Gamma \vec{f}(i), \qquad \vec{s}(i+1) \geq \vec{0} \tag{3}$$

The schedule $aababc$ of Figure 1 for example corresponds to the following sequence of state transitions:

$$\vec{s}(0) \ldots \vec{s}(6) = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \begin{bmatrix} 2 \\ 0 \end{bmatrix} \begin{bmatrix} 4 \\ 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} 3 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Inspecting the top most elements of the state vectors shows that the minimum buffer capacity on $c0$ is 4, and inspecting the bottom elements reveals that a buffer capacity of 2 suffices for $c1$. Depending on how buffer space is allocated to channels we can now draw two conclusions. Firstly, if all buffers share a *common* area of memory, the maximum buffer capacity required is 4, which is reached by states 2 and 4. Secondly if each channel has a *separate* buffer, the maximum buffer capacity is 6, since the maximum capacity of 4 for $c0$ is reached at state 2 and the maximum buffer capacity of 2 for $c1$ is reached at state 5.

We now review those results from the literature about the semantics of SDF that we need in the sequel.

An SDF graph is *consistent* iff $\text{rank}(\Gamma) = N - 1$ [13]. A deadlock free and consistent SDF graph has periodic schedules [8].

The N element *repetition vector* $\vec{r}$ is the least non-trivial integer solution of the equation [13]:

$$\Gamma \vec{r} = 0 \tag{4}$$

The repetition vector for Figure 1 is $\vec{r} = [3 \ 2 \ 1]^T$.

Assume that for a given channel x the production rate is p, the consumption rate is c, and the initial number of tokens on the channel is t, the *lower bound* on the buffer capacity of the channel for a deadlock free schedule is [2]:

$$\text{lwb}_c(x) = p + c - d + t \bmod d, \quad \textbf{where } d = \gcd(p, c) \tag{5}$$

Assume that for a given channel x the production rate is p, the channel is connected to the output port of node n, and the component of the repetition vector corresponding to node n is r, the *upper bound* on the buffer capacity of the channel for a deadlock free schedule is [2]:

$$\text{upb}_c(x) = r \times p \tag{6}$$

```
byte c0, c1;  /* Common buffer pool model */
init{ do
/*a*/ ::                 c0+=2;
/*b*/ :: (c0>=3) -> c0-=3; c1+=1;
/*c*/ :: (c1>=2) -> c1-=2;
      od }
/* LTL feasible:   [](c0+c1<=4) */
/* LTL infeasible: [](c0+c1<=3) */
```
```
byte c0, c1;       /* Separate buffer model */
byte s0=4, s1=2;
#define max(a,b) (a>b->a:b)
init{ do
/*a*/ ::                 c0+=2;        s0=max(c0,s0);
/*b*/ :: (c0>=3) -> c0-=3; c1+=1; s1=max(c1,s1);

/*c*/ :: (c1>=2) -> c1-=2;
      od }
/* LTL feasible:    [](s0+s1<=6) */
/* LTL infeasible: [](s0+s1<=5) */
```

Fig. 3. GBS model of the simple SDF graph with a common buffer pool (above) and separate buffers for each channel (below).

A lower bound on the buffer space for the whole graph is $\Sigma_{1 \le x \le C}\mathsf{lwb}_c(x)$ and an upper bound is $\Sigma_{1 \le x \le C}\mathsf{upb}_c(x)$.

With these results, a significant part of the problem of finding a periodic schedule with a minimum buffer size has been solved, because we can check first whether a graph is consistent. If a graph is indeed consistent, calculating the repetition vector gives the number of times each node must fire, and calculating the lower and upper bound on the buffer capacity we have the range in which to search for the minimum buffer size. Unfortunately, in practical cases the upper bound is typically much larger than the lower bound (See Table III). On the other hand, the lower bound is often also the minimum buffer size, which suggests that a good heuristic would be to look for a periodic schedule with the lower bound first. If this fails, a more general search is needed.

IV. MODEL CHECKING WITH SPIN

A state based model checker such as SPIN [11] is a tool that explores all possible behaviours of a Labelled Transition System, either to prove the absence of unwanted behaviour (safety properties), or to prove the existence of desired behaviour (liveness properties). As observed by GBS, when given an appropriate model of an SDF graph, the model checker can be used to check whether or not a schedule exists, calculating both the schedule and the minimum buffer size of each channel.

There are several reasons for choosing SPIN for our analysis. Firstly, SPIN is arguably one of the most powerful explicit state model checkers available. Secondly, the SPIN c_code extensions allow us to implement the Branch and Bound extensions of Section VII. Finally, as GBS also use SPIN, the comparison between GBS and our work is fair.

A. GBS models with a common buffer pool

We will describe the essence of the GBS models (Figure 3), indicating the direct correspondence between the model and the semantics of Section III. The state of the model consists of the pair of channel counters (i.e counting the number of tokens in each channel) c0 and c1. This pair represents the state vector of Equation 1. The do ... od statement causes the system to make a sequence of state transitions, and each guarded command :: ... corresponds to firing one of the nodes, provided that the command is enabled (i.e when the guard is true). The guards ensure that the state vector remains non-negative, as specified by condition of Equation 3. The assignments in each guarded command correspond to Equation 3. If more than one guard is true a non-deterministic choice is made to select one of the guarded commands. This selection corresponds to the non-deterministic choice of Equation 2.

The model of Figure 3 (above) is used to check whether the total amount of buffer space (i.e. when one common pool of buffer space is used for all channels) is less than or equal to 4. When presented to SPIN, the Linear Temporal Logic (LTL) property [](c0+c1<=4) requests the model checker to find a schedule represented as an infinite sequence of states, where each state satisfies (c0+c1<=4), the buffer capacity invariant. (In SPIN jargon the schedule represents a counter example to the error behaviour specified by the LTL formula). The model can also be used to verify that no periodic schedule exists with a bound less than or equal to 3 (using the second, infeasible property), thus proving that 4 is indeed the minimum size of the common buffer pool.

To avoid clutter, we show a simplified version of the GBS models. In particular all guarded commands :: ... in our models should be interpreted as atomic statements, i.e. they should be read as :: atomic{ ... }.

B. GBS models with separate buffers

The state space generated by SPIN from the model of Figure 3 (above) coincides with the state space of the SDF semantics as discussed in Section III, and may therefore be considered a good concrete model. The GBS model for the case where instead of one buffer pool, each channel has its own buffer space is shown in Figure 3 (below). The two variables s0 and s1 store the maximum number of tokens buffered by c0 and c1. GBS show that the *lower bound* optimisation, which initialises s0 and s1 to the lower bound calculated according to Equation 5 is effective. The reason is that if s0 and s1 are initialised to 0, a first set of *transient* states must be explored until s0 reaches 4 and s1 reaches 2. Then, the values of s0 and s1 must be maintained while a second set of *periodic* states is explored that represent the schedule. Since the schedule consists of the periodic set, it is beneficial to avoid the transient set. This is exactly what the GBS optimisation *lower bound* achieves.

The model of Figure 3 (below) can be used to check that the sum of the bounds on two separate buffers is 6 (feasible property), and that no periodic schedules are possible with a sum less than or equal to 5 (infeasible property).

```
byte na, nb, nc;    /* Same for both models */
#define c0 (na*2-nb*3)
#define c1 (nb*1-nc*2)
init{ do
/*a*/ :: (na<3)            -> na++;
/*b*/ :: (nb<2 && c0>=3) -> nb++;
/*c*/ :: (nc<1 && c1>=2) -> nc++;
      od }
#define r  (c0==0 && c1==0)
#define p0 ((c0<=3) && (c1<=2))
#define p1 ((c0<=4) && (c1<=1))
                        /* Common buffer model */
/* LTL feasible:   X ((c0+c1<=4) U r) */
/* LTL infeasible: X ((c0+c1<=3) U r) */
                        /* Separate buffer model */
/* LTL feasible:   X ((c0<=4 && c1<=2) U r) */
/* LTL infeasible: X (p0 U r) || X (p1 U r) */
```

Fig. 4. Our model (also showing the Limiting optimisations) of the simple SDF graph with a common buffer pool (middle) or separate buffers for each channel (below). The top part is common to both models.
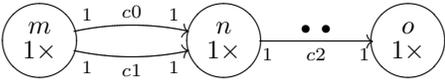

Fig. 5. Avoiding the transient is incomplete.

V. OPTIMISATIONS

Optimisations avoid searching those parts of the state space that cannot lead to periodic schedules, or that lead to schedules worse than we have already seen. An optimisation that may miss correct periodic schedules satisfying a given buffer constraint is *incomplete*. An optimisation that that may yield incorrect schedules is *unsound*. All types of optimisation may be useful. For example a schedule found by an incomplete optimisation is correct but it may be sub-optimal, and it is often possible to check via some alternative means whether a schedule found by an unsound optimisation is correct or not. We give examples of optimisations, indicating whether the optimisation is effective, sound and/or complete on a benchmark of commonly used SDF graphs, including some realistic applications.

A. Node counters

The GBS *channel* counters contain redundancy that can be avoided by using *node* counters instead (Figure 4). It is easy to calculate the value of a channel counter from the relevant node counters (as shown by the macro definitions for c_0 and c_1) but it is not possible the other way round. Unlike the channel counters, node counters are in principle unbounded, and should be used in combination with the Limiting optimisation (Section V-D). Sound. Complete. It depends on the SDF graph whether node counters are effective.

B. Avoiding the transient

GBS models produce schedules with a transient and a periodic part. For example the SDF graph of Figure 1 may generate a schedule such as $aa(babaac)*$ with a transient aa and a periodic part $(babaac)*$. Often there is a shorter schedule without a transient part. To avoid a schedule with a transient we use an LTL property that ensures that the schedule begins *and* ends in the initial state. This property is of the general form X (p U r) with the following interpretation. Assume that in the initial state property r (characterising the initial state) is true. The neXt operator X moves to the next state. Then we use the Until operator U to specify a sequence of states for which the property p (the buffer constraint) holds, until finally again the property r holds (and also p since r implies p). Using the feasible LTL property of Figure 4 (middle) we can verify that a periodic schedule exists with a bound of 4 on the common buffer pool. To verify that no schedules exist for smaller bounds the infeasible property of Figure 4 (middle) can be used. For this particular benchmark, as we argued in Section III, 4 is provably the lower bound on the common buffer size. Therefore, there is no need to run the model checker to confirm that 4 is indeed the minimum bound. The only benchmarks where the lower bound is not the minimum bound are ade and adebetter (See Section VI for more information about the benchmarks).

Avoiding the transient is sound (because we are not changing schedules) but incomplete as demonstrated by the example of Figure 5, which admits a schedule $o(omn)*$ with a common buffer of size 2, that is found by the GBS model but not by our model. (Avoiding the transient is complete for the separate buffer case.) Effective.

Our model for the separate buffer pool (Figure 4 below) is the same as for the common buffer pool. Unfortunately we need an LTL property that is exponential in size in the number of channels, which is clearly infeasible. Instead, we use a property that consists of as many conjunctions as there are channels, with each conjunct reducing the buffer space for its channel by 1. Incomplete.

C. Priority

When making a non-deterministic choice, SPIN explores the guards top down, so reversing the order of the guards causes different parts of the state space to be explored first. This property of the semantics of SPIN makes it possible to model the priority principle [1, Section 4.1], which gives increasing priority to successive nodes in a chain. Most practical graphs, including the graphs in our benchmark are not chains but cyclic graphs, where the priority principle cannot be applied. Indeed the benchmark results show no significant changes when applying the principle. Sound and Complete, because we are merely changing the order in which schedules appear. Ineffective.

D. Limiting

The number of times a node fires is limited by the repetition vector (Equation 4), because a periodic schedule must invoke each node at least as often as given by the repetition vector. This is shown by the guarded commands of Figure 4, where each guarded command has a condition for the form
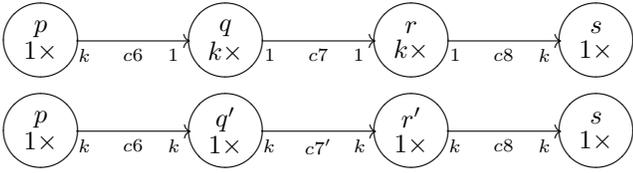
Fig. 6. Chain from the h263 decoder (above) and the same chain (below) without spurious repetition of the nodes q and r, where $k = 2376$.


```
#define p4  (c0<=5&&c1<=5&&c2<=1&&c3<=1&&c4<=3)
#define p2  (c0<=5&&c1<=5&&c2<=3&&c3<=1&&c4<=1)
#define r   (c0==0&&c1==0&&c2==0&&c3==0&&c4==0)
/* LTL sound:   X (p2 U r) */
/* LTL unsound: X (p4 U r) */
```

Fig. 7. Two chains with a common start node h, and a common end node l in the state where node h has fired three times, i twice and j once. A sound and an unsound version of the LTL property are shown below.

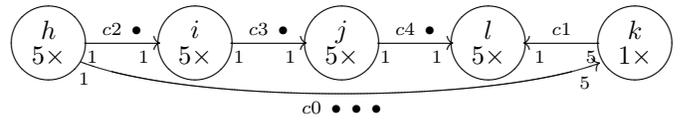nx<y. Sound, because we are not changing any schedules. Incomplete as illustrated in Figure 5.

E. Clustering

In realistic data flow graphs the firing rate of some nodes may differ considerably. Figure 6 (above) gives an example of a chain from the h263 decoder, where nodes p and s are fired once against k times for nodes q and r. This difference in firing rate increases the number of interleavings exponentially in k (as the Catalan number of k) and hence also increases the size of the state space. Our clustering optimisation *transforms* a chain into one with smaller differences in the firing rates such as Figure 6 (below). To transform nodes q and r into q' and r' the consumption and production rates of these nodes are multiplied by k, at the same time the entries in the repetition vector of the nodes are divided by k. Let Γ and Γ' be the topology matrix before and after the transformation. It is easy to check that $\mathsf{rank}(\Gamma) = \mathsf{rank}(\Gamma')$ hence the transformation does not affect consistency. However, the transformation is unsound, since $\mathsf{lwb}_c(c7') = k$ whereas $\mathsf{lwb}_c(c7) = 1$.

Once a schedule has been found for the transformed model, it is possible to construct a schedule for the original model. For example given a schedule $(pq'r's)^*$ for the transformed system, we can represent q' in the schedule of the transformed system by q^k in the original system, and likewise for r'; this yields a schedule $(pq^k r^k s)^*$ for the original system. Unfortunately, this schedule requires a buffer of size k for channel $c7$. We can do better than this by interleaving q and r, which yields the following schedule for the original system: $(p(qr)^k s)^*$. Using Equation 3 it is possible to prove (simply by replaying the schedule) that this is indeed a valid schedule for the original system of Figure 6 (above). Using Equation 5 we can also prove that this is an optimal schedule. Sound if we include the transformation of the schedule as suggested above. Incomplete. Effective.

F. Look ahead

Look ahead is an optimisation where each node has knowledge of the behaviour of its immediate successors. Look ahead permits a node to fire only when at least one of its outputs has insufficient tokens for the successor node. Consider the example of Figure 7. Node h may fire when its successor i has insufficient input or when its successor k has insufficient input (or both). The idea behind look ahead by node h is that if both successors do have sufficient input, h is blocked to avoid overfilling $c0$ and $c2$.

TABLE II
SUMMARY OF THE OPTIMISATIONS. * = IF THE TRANSFORMATION OF THE SCHEDULE IS INCLUDED

Optimisation	Effective	Sound	Complete
Node counters V-A	±	+	+
Avoid transient V-B	+	+	-
Priority V-C	-	+	+
Limiting V-D	+	+	-
Clustering V-E	++	-	+*
Look ahead V-F	+	+	-

The example of Figure 7 has been constructed such that there are two chains (i.e. the upper chain h, i, j, and l and the lower chain h, k, and l). Both the lower chain and the upper chain have to store 5 tokens. However, looking at the production and consumption rates of the upper chain alone it would appear that one token on each channel (hence a total of 3) would suffice on the upper chain.

There are many ways in which to distribute the two extra tokens over the buffer capacity of the upper chain. For example property p4 of Figure 7 forces the excess to be stored in $c4$, and property p2 stores the excess in $c2$. However, only one of these methods (i.e. property p2) is compatible with the optimisation for look ahead. To illustrate this point Figure 7 shows the state of the system where node h has fired three times, i twice and j once. The entire system is now blocked: Nodes k and l are blocked because there are *insufficient* tokens on their input channels and nodes i and j are blocked because there are already *sufficient* tokens on their output channels. Node h is blocked because p4 only allows the excess to be stored in $c4$. If on the other hand we would have used property p2, the network would not have been blocked. Incomplete. In the adebetter benchmark, which has been carefully constructed by Adé to demonstrate the intricacies of SDF scheduling [2], look ahead will increase the minimum buffer capacity by one. Unsound. Effective.

VI. CHECKING THE OPTIMAL BUFFER SIZE

Our benchmark consists of 10 SDF graphs taken from various sources. The benchmarks simple, bipart, cddat, modem, ade, adebetter, inmarsat, and h263 are used by GBS [7], the benchmarks mp3sys and mp3dec are used by Stuik et al [17]. These benchmarks are used by many other authors in the field

TABLE I

STATES STORED BY SPIN FOR THE BEST VERSIONS OF THE 10 BENCHMARKS AND MS EXECUTION TIME FOR STATE OF THE ART RESEARCH TOOLS. (* = ADEBETTER WITHOUT THE LOOK AHEAD OPTIMISATION, † ENTRIES SWAPPED IN GBS [7, TABLE 1])

	simple	bipart	cddat	modem	ade	adebetter	inmarsat	V-E	h263	V-E	mp3dec	mp3sys	V-E
	States stored checking feasible schedule with given bound. Separate buffer space												
GBS	11	88	4127	210	497	8602†	2862	66	4758	9	139	19308	1797
V-D, V-F	8	84	614	50	47	129*	1133	52	4758	8	15	16385	125
	States stored checking infeasible schedule with given bound. Separate buffer space												
GBS	2	2	2	2	2241	1708†	2	2	2	2	2	2	2
V-D	-	-	-	-	581	721	-	-	-	-	-	-	-
	States stored checking feasible schedule with given bound. Common buffer space												
GBS	9	124	755	1231	366	156	180369	163	9511	11	23	55004	424
V-D, V-F	8	94	614	176	96	109	1110	52	4758	8	15	16381	121
	States stored checking infeasible schedule with given bound. Common buffer space												
GBS	4	150	3542	853	303	140	aborted	240	aborted	7	12	aborted	925
V-D	3	149	3541	852	127	139	13102300	81	2826250	4	8	19653500	925
	milliseconds CPU time±standard deviation checking feasible+infeasible schedule. Separate buffer space												
SDF3	6±4	6±4	8±4	10±4	42±4	8±3	19±4	11±5	22±4	7±5	10±3	55±5	7±4
Hebe	11±3	11±3	11±3	15±3	12±3	12±3	16±3	15±3	12±3	12±4	14±2	11±3	12±3
SPIN	18±8	20±8	20±9	19±8	41±16	41±19	24±9	18±9	33±10	18±9	19±10	70±12	19±8

and are therefore assumed to be representative for SDF graphs.

To avoid creating the same variants of 10 different benchmarks, we wrote a C program that given the topology matrix and the initial token assignment of an SDF benchmark generates the SPIN models necessary to explore the optimisations described in Section V and summarised in Table II.

Table I shows for each benchmark the best results that we obtain in terms of the number of states stored by SPIN to find a feasible schedule, or to prove that such a schedule does not exist. In all cases the same or a better feasible schedule is found, indicating that in the benchmark examples unsound and incomplete optimisations do not cause problems.

The table is divided into five sections. The first two sections report the states stored for models where each channel has a separate buffer. The next two sections apply to models where there is one common buffer pool for all channels. The first and third section presents the results when looking for (the first occurrence of) a feasible schedule, whereas in the second and fourth section we report on the number of states encountered when exploring the state space exhaustively because there is no feasible schedule. The last section reports execution times.

The rows marked GBS in the first column are the best results of GBS taken from their paper [7, Table 1]. We have repeated the experiments of GBS to be able to include GBS results on the mp3dec and mp3sys benchmarks also.

In the entries marked "aborted" we terminate the experiment after 5 minutes of CPU time. In all benchmarks except ade and adebetter a feasible schedule is found with the minimum buffer size, so it does not make sense (indicated by a hyphen) to try to prove that a configuration with less buffer space than the theoretical minimum is infeasible.

The rows *not* marked GBS represent our best results, indicating which optimisation(s) have been deployed (referred to by section number). Without exception, our results are better than GBS, in some cases by several orders of magnitude, for example in case of the inmarsat benchmark. Overall, the most important cause for the improvement is the use of node

counters with the Limiting optimisation instead of channel counters.

The benchmarks with large differences in production and consumption rates on the same channel, such as inmarsat, h263 and mp3sys benefit significantly from the clustering optimisation, by up to five orders of magnitude. The columns marked V-E report the data for the clustered versions of these benchmarks. The reason is that the number of interleavings is exponential (the Catalan number) in the number of times each node may fire. The clustering optimisation reduces this to a linear dependency, hence the significant difference.

State of the art research tools do not provide an equivalent to the number of states explored as a metric. Therefore, to compare our results to those tools, we have repeated the first (i.e. Separate buffer, feasible+infeasible schedule) experiment for all benchmarks using SDF3 [18] and Hebe [19], all on the same Linux machine. The SPIN models and SFD3 provide an exact solution, Hebe calculates a good approximation (within 10%) to the minimum buffer size. The SPIN models can only be used to analyse the minimal buffer capacity for deadlock-free execution of SDF graphs, whereas SDF3 and Hebe can also take throughput into account. We have tried to make sure that this does not give our approach an unfair advantage; in fact the authors of SDF3 have helped us to make various modifications to avoid bias as much as possible. The CPU user times measured as an average over 50 runs as well as the sample standard deviation are shown in the last section of Table I. The error margins overlap so much that we conclude that the performance of all three tools is comparable. This shows that it is cost effective to gain insights by experimenting with a range of optimisations using a general purpose tool, before undertaking costly special purpose tool development. For example GBS spent only a few days implementing the minimum buffer size algorithm of the SDF3 tool (which computes the entire buffering-throughput trade-off space), after having spent more time experimenting with SPIN. Ultimately, an ideal tool framework would include

TABLE III

	simple	bipart	cddat	modem	ade	adebett	inmarsV-E	h263V-E	mp3dec	mp3sysV-E
	Bounds for the common buffer case based on the analysis of nodes.									
$s = \min_{1 \leq x \leq N} \mathsf{lwb}_n(x)$	2	10	1	1	5	4	240	3	1	5
$g = \max_{1 \leq x \leq N} \mathsf{lwb}_n(x)$	4	16	15	10	25	9	720	4752	2	1536
$\min_n$	4	26	16	13	67	18	1008	4754	2	1539
$\mathsf{upb}_n = \Sigma_{1 \leq x \leq N} \mathsf{lwb}_n(x)$	12	60	60	149	105	72	5472	23762	22	8843
state stored ratio	1.0	1.0	1.9	1.5	3.9	5.0	3.9	1.4	1.0	1.2
SPIN runs	1	2	2	4	10	4	3	2	1	2
	Bounds for the separate buffer case based on the analysis of channels.									
$\Sigma_{1 \leq x \leq C} \mathsf{lwb}_c(x)$	6	28	32	38	49	39	3072	9508	12	2961
$\min_c$	6	28	32	38	83	42	3072	9508	12	2961
$\Sigma_{1 \leq x \leq C} \mathsf{upb}_c(x)$	8	264	1021	61	209	133	3936	9508	12	27406

a range of techniques [9].

VII. FINDING THE OPTIMAL BUFFER SIZE

Thus far we have explored optimisations to the GBS approach to check whether a given bound on the buffer size is optimal. The check requires running the model checker twice: once to verify that a schedule with the given bound can be found, and a second time to verify that no schedule can be found with a bound of one less. Finding the optimal bound is a more challenging problem for two reasons. Firstly, we must be able to calculate an initial guess for the minimum bound. Secondly, depending on the quality of the guess, we may have to run the model checker several times. To make the problem even more challenging, we will study the case of the common buffer, which as Table I shows, requires considerable more work (i.e. more states to be stored) than the separate buffer case. Therefore in this section we will develop the necessary theory and apply the theory in practical optimisations to find optimal bounds on common buffers for the benchmarks.

A. Theoretical lower bound

The literature provides theoretical results on the lower bound and upper bound on the buffer space required for SDF graphs when each channel buffer resides in a separate area of memory (c.f. $\mathsf{lwb}_c(.)$ and $\mathsf{upb}_c(.)$ in Section III). Unfortunately, we have not been able to find equivalent results for the case where all buffers share a common area of memory. Therefore, we will develop new theory to calculate a lower bound on the total common buffer space required by an SDF graph. The idea for the calculation is to analyse each node n separately by decoupling n from the graph, together with all its direct neighbours and the channels connecting n to the neighbours. We will call this sub graph the decoupled graph of n. For example, decoupling node a in Figure 1 would create a new graph consisting of copies of nodes a, and b, and the connecting channel $c0$. Decoupling node d in Figure 2 would create a new graph consisting of a copy of node d, and *two* copies of node e as well as the connecting channels $c0$, and $c1$. The schedule admitted by a decoupled graph of node n is completely unconstrained, hence the schedule is defined by the following algorithm:
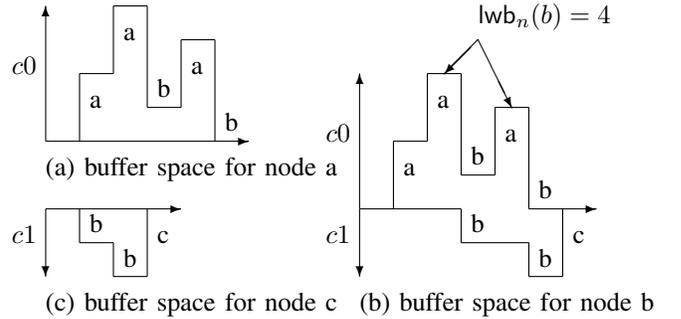

Fig. 8. Common buffer space analysis for SDF graph from Figure 1

1. put the initial tokens on all channels of the decoupled graph of n.
2. repeat
2.1 Fire each node sending tokens to n as often as necessary to satisfy the consumption rates of the inputs to node n.
2.2 Fire node n once.
2.3 Fire each node receiving tokens from n as often as possible.
3. until node n has been fired $\vec{r}(n)$ times.

The lower bound on the total common buffer size $\mathsf{lwb}_n(n)$ of the decoupled graph for node n is then the maximum number of tokens on all channels of the decoupled graph observed during the execution of the algorithm.

For example the total buffer capacity for the decoupled graph of node a from Figure 1 is $\mathsf{lwb}_n(a) = 4$. The maximum is reached after two firings of a as shown in Figure 8(a). Figure 8(b,c) show that $\mathsf{lwb}_n(b) = 4$, and $\mathsf{lwb}_n(c) = 2$.

To prove that $\mathsf{lwb}_n(n)$ is indeed a lower bound on the amount of common buffer space required by node n we analyse the algorithm. Line 2.1 ensures that when node n fires, no more tokens are present on the input channels to node n than strictly necessary to satisfy the consumption rates of n. In a realistic schedule, there may be more tokens present on the input channels than in the decoupled graph, but not less. Line 2.3 ensures that the output channels are emptied as much as possible. In a realistic schedule there may be more tokens that

Initial guess g	States stored	Feasible bounds
10	30	none
14	66	none
18	157	none
22	7648	21,20,19,18
total	7901	
19	1568	18
ratio	5.0	

remain in the output channels than in the decoupled graph, but not less. Summarising, both on the input and on the output side of node n no more tokens are present than strictly necessary. Hence $\mathsf{lwb}_n(n)$ gives a lower bound on the amount of common buffer space required by node n.

The complexity of the algorithm to calculate $\mathsf{lwb}_n(n)$ is linear in $\vec{r}(n)$.

B. Optimisations for the minimum bound

Equipped with a lower bound on the size of the common buffer pool for each node we are ready to develop a scheduling algorithm. A good basis for this is the SPIN version of the Branch and Bound algorithm as proposed by Ruys [15], which can be adapted to our needs as follows:

1. Start with an initial guess g for the optimal bound and a step size s, where: $g \leftarrow \mathsf{max}_{1 \leq n \leq N}\mathsf{lwb}_n(n)$, and $s \leftarrow \mathsf{min}_{1 \leq n \leq N}\mathsf{lwb}_n(n)$.
2. repeat
2.1 Let SPIN find a schedule with an optimal bound $b \leq g$.
2.2 if such a schedule can be found then exit
2.3 else $g \leftarrow g + s$
3. end repeat

Since g is a lower bound on the buffer size, and $s > 0$, the algorithm is guaranteed to terminate. The SPIN models used are basically the same as the GBS models, with the modifications described by Ruys [15] to find the minimum bound $b \leq g$. The appendix provides the complete source code of the simple benchmark. Note that to check whether an optimal bound exists for guess g, we initialise with $g + 1$ (see Table IV), to let SPIN find a bound less than $g + 1$, i.e. g.

To analyse how successful the Branch and Bound strategy is, we take as an example the adebetter benchmark. Table IV shows that starting with an initial guess of $g = 10$, after visiting 30 states SPIN terminates because no feasible schedules can be found with a bound lower than 10. Then the guess is increased by step size $s = 4$ to 14, and SPIN is run a second time, again without finding a schedule. This is repeated until $g = 22$. Now SPIN finds a feasible schedule with a bound of $b = 21$, and starts looking for another schedule with a bound lower than 21. Indeed such a schedule is found; with a bound of 20 etc until a schedule with a bound of 18 is found, and no schedule can be found with a bound lower than 18. The total number of states stored (7901) is a measure for the amount

of work performed to search for a feasible schedule with the optimal bound.

The choice of the initial guess g, and the step size s is critical for the efficiency of the search. For many benchmarks, the initial guess is a reasonable bound, as we can see by comparing the second row (labelled $g = \mathsf{max}_{1 \leq x \leq N}\mathsf{lwb}_n(x)$) and the third row (labelled min_n) that shows the true minimum bound in table III for all benchmarks. For completeness the table also shows the step size $s = \mathsf{min}_{1 \leq x \leq N}\mathsf{lwb}_n(x)$ and a (poor) upper bound calculated as $\Sigma_{1 \leq x \leq N}\mathsf{lwb}_n(x)$.

The choice of the step size is motivated as follows. The initial guess represents the needs of the decoupled graph with the largest buffer requirements, and the step size represents the needs of the decoupled graph with the smallest buffer requirements. In the extreme case of an SDF graph with only two nodes, the optimal buffer size can be anywhere between g (when the buffer capacities of the two nodes completely overlap) and $g + s$ (when the buffer capacities are completely disjoint). So if the optimal buffer is not found with the initial guess g it will definitely be found with the next guess $g+s$. In an SDF graph with more than 2 nodes, the step size controls how many more iterations than two could be necessary. There are two reasons why starting with an initial guess that is likely to be too low and increasing the guess is better than starting with an initial guess that is too high. Firstly, there are many schedules with a sub optimal buffer size, such that the search starting from a high initial guess yields many spurious results that are time consuming to find and discount. Secondly, an initial guess that is too low causes many branches in the search space to be pruned quickly.

To indicate how good the search optimisations are, Table IV (bottom) shows that for adebetter with an initial guess $g = 19$ (i.e. one more than the true lower bound) the number of states visited is 1568. This means that to find the best schedule SPIN has to do about 5 times as much work as to check the best schedule. Table III shows these work ratios for each benchmark (row labelled state stored ratio) as well as the relevant bounds. The conclusion is that with our Branch and Bound algorithm finding the minimum schedule on the benchmark is up to five times more expensive than checking the best bound, which we believe is a good result.

VIII. CONCLUSIONS AND FUTURE WORK

Many authors have used model checkers to solve scheduling problems [3] [4] [5] [6] [12] [16] [10], but Geilen, Basten and Stuijk [7] (GBS) were the first to use SPIN for the analysis of SDF graphs. Their results are promising but inconclusive in the sense that some realistic SDF graphs cannot be analysed effectively. Our approach towards *checking* given bounds using unsound and incomplete optimisations generally pays off and in specific cases exponential complexity is reduced to linear complexity by our clustering optimisation. As a result all case studies used can be analysed by SPIN in about the same time as needed by state of the art research tools. This makes SPIN a useful prototype tool for the buffer size analysis of SDF

graphs. In the end the most effective techniques could then be integrated in special purpose tools such as SDF3 and Hebe.

We offer new theory and an efficient Branch and Bound algorithm to *find* minimum bounds, thus solving a problem not considered by GBS. The main advantage of using SPIN as the Swiss army knife of computer science is that no special purpose tools have to be created in order to gain deep insight into NP complete problems by extensive experimentation with optimisations. It would be an interesting challenge to extend the SPIN models, particularly with throughput constraints, or to more liberal dataflow models, such as models with data-dependent rates. Furthermore, we will investigate whether the Branch and Bound optimisations can be further improved, e.g., by using binary search, or by looking ahead in the search path.

REFERENCES

[1] M. Adé. *Data Memory Minimization for Synchronous Data Flow Graphs Emulated on DSP-FPGA Targets*. PhD thesis, Katholieke Universiteit Leuven, Oct 1996.

[2] M. Adé, R. Lauwereins, and J. A. Peperstraete. Data memory minimisation for synchronous data flow graphs emulated on DSP-FPGA targets. In *34th Design Automation Conf. (DAC)*, pages 64–69, Anaheim, California, Jun 1997. IEEE Computer Society.

[3] Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim Larsen, Paul Pettersson, and Judi Romijn. Efficient Guiding Towards Cost-Optimality in UPPAAL. In Tiziana Margaria and Wang Yi, editors, *Procs. of the 7th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *LNCS*, pages 174–188, Genova, Italy, April 2001. Springer.

[4] Ed Brinksma and Angelika Mader. Verification and Optimization of a PLC Control Schedule. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN Model Checking and Software Verification, Procs. of the 7th Int. SPIN Workshop (SPIN'2000)*, volume 1885 of *LNCS*, pages 73–92, Stanford, California, USA, August 2000. Springer.

[5] Ansgar Fehnker. Scheduling a Steel Plant with Timed Automata. In *Procs. of the 6th Int. Conf. on Real-Time Computing Systems and Applications (RTCSA 1999)*, pages 280–286. IEEE Computer Society, 1999.

[6] Ansgar Fehnker. *Citius Vilius Melius – Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. PhD thesis, University of Nijmegen, The Netherlands, April 2002.

[7] M. C. W. Geilen, T. Basten, and S. Stuijk. Minimising buffer requirements of synchronous dataflow graphs with model checking. In *42nd Design Automation Conf. (DAC)*, pages 819–824, San Diego, California, Jun 2005. ACM.

[8] A. H. Ghamarian, M. C.W. Geilen, T. Basten, B. D. Theelen, M. R. Mousavi, and S. Stuijk. Liveness and boundedness of synchronous data flow graphs. In *6th Int. Conf. on Formal Methods in Computer Aided Design (FMCAD)*, pages 68–75, San Jose, California, Nov 2006. IEEE Computer Society Press.

[9] R. Govindarajan, G. R. Gao, and P. Desai. Minimizing buffer requirements under Rate-Optimal schedule in regular dataflow networks. *J. VLSI Signal Process. Syst.*, 31(3):207–229, Jul 2002.

[10] S. Haynal and F. Brewer. Representing and scheduling looping behavior symbolically. In *Int. Conf. on Computer Design*, pages 552–555, Austin, Texas, Sep 2000. IEEE Computer Society.

[11] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference manual*. Pearson Education Inc, Boston Massachusetts, 2004.

[12] Kim Larsen, Gerd Behrmann, Ed Brinksma, Ansgar Fehnker, Thomas Hune, Paul Pettersson, and Judi Romijn. As Cheap As Possible: Efficient Cost-Optimal Reachability for Priced Timed Automata. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Procs. of the 13th Int. Conf. on Computer Aided Verification (CAV 2001)*, volume 2102 of *LNCS*, pages 493–505, Paris, France, July 2001. Springer.

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

[14] P. K. Murthy, S. S. Bhattacharyya, and E. A. Lee. Joint minimization of code and data for synchronous dataflow programs. *Formal Methods in System Design*, 11(1):41–70, Jul 1997.

[15] T. C. Ruys. Optimal scheduling using branch and bound with SPIN 4.0. In T. Ball and S. K. Rajamani, editors, *10th Int. SPIN Workshop on Model Checking Software*, volume 2648 of *LNCS*, pages 1–17, Portland, Oregon, May 2003. Springer.

[16] Theo C. Ruys and Ed Brinksma. Experience with Literate Programming in the Modelling and Validation of Systems. In Bernhard Steffen, editor, *Procs. of the 4th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, volume 1384 of *LNCS*, pages 393–408, Lisbon, Portugal, April 1998.

[17] S. Stuijk, M. C. W. Geilen, and T. Basten. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *43rd Design Automation Conf. (DAC)*, pages 899–904, San Francisco, California, Jul 2006. ACM.

[18] S. Stuijk, M. C.W. Geilen, and T. Basten. SDF3: SDF for free. In *6th Int. Conf. on Application of Concurrency to System Design (ACSD)*, pages 276–278, Turku, Finland, Jun 2006. IEEE Computer Society.

[19] M. H. Wiggers, M. J. G Bekooij, P. G. Jansen, and G. J. M. Smit. Efficient computation of buffer capacities for Multi-Rate Real-Time systems with Back-Pressure. In *4th Int. Conf. on Hardware/Software Codesign an System Synthesis (CODES+ISSS)*, pages 10–15, Seoul, Korea, Oct 2006. ACM.

IX. Appendix

The complete source code of the simple benchmark, which, starting from an initial guess of 5 lowers `__best` each time a schedule is found with a better bound. The assignment `first=false` in UPDATE can be optimised away at the expense of a longer and less readable model.

```
c_state "int __best = 5" "Hidden"

#define MAX(a,b)  (a>b->a:b)
#define SUM       (ch[0]+ch[1])
#define WORSE     (c_expr{(now.maxsum)>=__best})
#define UPDATE    first=false; \
                  maxsum=MAX(maxsum,SUM)

#define PRODUCE(c,n) ch[c] = ch[c] + n
#define CONSUME(c,n) ch[c] = ch[c] - n
#define WAIT(c,n)    ch[c]>=n

byte ch[2], maxsum;
bool first=true;

init{
end:
   do
   :: atomic{
      (!first&&(ch[0]==0&&ch[1]==0))->break;
   }
/* Actor_c */
   :: atomic{
      WAIT(1,2) ->
      CONSUME(1,2);
      UPDATE;
   }
/* Actor_b */
   :: atomic{
      WAIT(0,3) ->
      CONSUME(0,3);
      PRODUCE(1,1);
      UPDATE;
   }
/* Actor_a */
   :: atomic{
      PRODUCE(0,2);
      UPDATE;
   }
   od;
   c_code{\
      if( now.maxsum < __best ) {\
         __best = now.maxsum;\
         printf( ">best now: %d\n",__best);\
         putrail();\
         Nr_Trails--;\
      }\
   };
}

never{ /* !<> WORSE */
accept_init:
   if
   :: (! (WORSE)) -> goto accept_init
   fi;
}
```

The bash script shown below runs SPIN iteratively, starting from the initial `guess`, and incrementing the `guess` by step, until a feasible schedule is found as indicated by the presence of a trail file. Note that that the verifier `pann.c` is compiled only once.

```
spin -a ${prom_file}

# add -#N option to pan to initialise __best
sed -e "/default : usage(efd); break;/i\
case '#': __best = atoi(&argv[1][2]);\
          break;" < pan.c > ppan.c

# note: ppan is now the verifier
gcc -o ppan -DSAFETY ppan.c

while [ ! -e "$trail_file" ]; do
   out_file=${prom_file}_${2}_${guess}.log
   echo "now try __k = ${guess}"
   echo "     file=${out_file}"
   time ./ppan -\#${guess} -c0 -E \
               -w24 -m100000 \
         > ${out_file} 2>&1
   guess=$((guess+step))
done
```

For a full explanation of the mechanisms used please consult Ruys [15].