Scheduling optimisations for SPIN
to minimise buffer requirements in synchronous data flow

Pieter H. Hartel and Theo C. Ruys
University of Twente, The Netherlands

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, and improve on 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 tools. The key to success is creating abstract SPIN models, using the semantics of SDF to prove when using (even unsound and/or incomplete) abstractions are justified. The main benefit of our approach lies in gaining deep insight in the optimisations at relatively low cost.

1 Introduction

Synchronous Data Flow (SDF) is a paradigm suitable for describing a class of Digital Signal Processing (DSP) applications [5]. 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 be 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 are used in Signal Processing to describe DSP and multimedia applications. A typical application is intended to 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.

SDF graphs come in many flavours; we focus on the classical variant as discussed by Lee and Messerschmitt [5].

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 problems are typically NP complete [6], the simple semantics of SDF makes it possible to prove a wealth of useful properties that can be used as optimisations in the analysis algorithms. However, designing the algorithms, and experimenting with the optimisations requires a significant amount of effort.

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 [3] (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 provide significant 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 3 . . . 6). Secondly, we develop new theory and the algorithms necessary for finding the minimum bounds (Section 7) for the common buffer case.

2 Examples

To give the intuition for the semantics of SDF we discuss three examples, the first of which is shown in Figure 1. 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. 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 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 \((aabaab)\) and \((aabaab)\). These schedules are actually sequential schedules. In the first schedule the data dependencies inhibit concurrency, in the sec-

\[
\begin{align*}
& a \quad 3 \times 2 \quad c0 \quad 3 \\
& b \quad 2 \times 1 \quad c1 \quad 2 \\
& c \quad 1 \times
\end{align*}
\]

Figure 1: Simple SDF graph with three nodes \(a\), \(b\), and \(c\) and two edges \(c0\) and \(c1\).
ond schedule a and b may fire concurrently: \((aa|bb)bc\)\(^*\). Following GBS, in the sequel we will focus on sequential schedules. The minimum buffer capacity for \(e0\) required by the second (sequential) schedule is 6 tokens, whereas for the first schedule 4 tokens would suffice on \(e0\). Therefore schedule \((aa|bb)bc\)\(^*\) is the best of the two schedules in terms of the buffer capacity for \(e0\).

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. Again infinitely many periodic schedules are possible, but this time there is only one shortest: \((eed)\)\(^*\). 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.

### 3 Semantics

An SDF graph with \(N\) nodes and \(C\) channels can be characterised completely 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 \(\Gamma\) for the SDF graph of 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:

\[
\vec{s}(0) = \begin{bmatrix}
0 \\
0
\end{bmatrix}
\]

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 \(\vec{f}(i)\) (of height \(N\)):

\[
\vec{f}(i) = \begin{bmatrix}
1 \\
0
\end{bmatrix} \text{ or } \begin{bmatrix}
0 \\
1
\end{bmatrix} \text{ or } \begin{bmatrix}
0 \\
0
\end{bmatrix}
\]

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:

\[
\vec{s}(i+1) = \vec{s}(i) + \Gamma \vec{f}(i), \quad \vec{s}(i+1) \geq \vec{0}
\]

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

\[
\vec{s}(0) = \begin{bmatrix}
0 \\
0 \\
0 \\
0 \\
0 \\
0
\end{bmatrix}, \quad \vec{s}(1) = \begin{bmatrix}
2 \\
0 \\
0 \\
1 \\
1 \\
0
\end{bmatrix}, \quad \vec{s}(2) = \begin{bmatrix}
4 \\
0 \\
0 \\
1 \\
1 \\
0
\end{bmatrix}, \quad \vec{s}(3) = \begin{bmatrix}
1 \\
1 \\
1 \\
1 \\
1 \\
0
\end{bmatrix}, \quad \vec{s}(4) = \begin{bmatrix}
3 \\
0 \\
0 \\
0 \\
0 \\
0
\end{bmatrix}, \quad \vec{s}(5) = \begin{bmatrix}
2 \\
0 \\
0 \\
0 \\
0 \\
0
\end{bmatrix}
\]

Inspecting the top most elements of the state vectors shows that the minimum buffer capacity on \(e0\) is 4, and inspecting the bottom elements reveals that a buffer of 2 suffices for \(e1\). 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 \(e0\) is reached at state 2 and the maximum buffer capacity of 2 for \(e1\) 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 \(\text{rank}(\Gamma) = N - 1\) \cite{5}. A consistent SDF graph has periodic schedules.

The \(N\) element repetition vector \(\vec{r}\) is the least non-trivial solution of the equation \cite{5}:

\[
\Gamma \vec{r} = \vec{0}
\]

The repetition vector for the example of 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 \cite{2}:

\[
lwb_e(x) = p + c - d + t \mod d, \quad \text{where } d = \gcd(p, c)
\]

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 \cite{2}:

\[
\upb_e(x) = r \times p
\]

The lower bound on the buffer space for the whole graph is \(\sum_{1 \leq x \leq C} lwb_e(x)\) and the upper bound is \(\sum_{1 \leq x \leq C} upb_e(x)\).

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 Figure 2). 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.

### 4 Model checking with SPIN

A state based model checker such as SPIN \cite{4} is a tool that explores all possible behaviours of a Labelled Transition system generated by a Promela model, either to prove the absence of unwanted behaviour.
is ef-

3

3

3

3

(above) has two problems. Firstly, the state space of this model

3

3

3

3

presents the essence of such a more abstract model. There are

(above) can be used to check that the sum of

1

1

1

1

variables (i.e counting the number of tokens in each channel)

(a 0 \rightarrow a b)

\text{do}

# Define max(a, b) (a > b \rightarrow a:b)

\text{init}{

/* LTL feasible: [\neg (c0+c1<=4)] */

/* LTL feasible: [\neg (c0+c1<=3)] */

/* LTL infeasible: [\neg (s0+s1<=5)] */

/* LTL infeasible: [\neg (s0+s1<=6)] */

/* LTL infeasible: [\neg (s0+s1<=5)] */

byte c0, c1; /* Common buffer pool model */

byte s0=4, s1=2;

#define max(a, b) (a>b->a:b)

/* LTL feasible: [\neg (c0+c1<=4)] */

/* LTL feasible: [\neg (c0+c1<=3)] */

/* LTL infeasible: [\neg (s0+s1<=5)] */

/* LTL infeasible: [\neg (s0+s1<=6)] */

byte c0, c1; /* Separate buffer model */

byte s0=4, s1=2;

#define max(a, b) (a>b->a:b)

/* LTL feasible: [\neg (c0+c1<=4)] */

/* LTL feasible: [\neg (c0+c1<=3)] */

/* LTL infeasible: [\neg (s0+s1<=5)] */

/* LTL infeasible: [\neg (s0+s1<=6)] */

Figure 3: The essence of the GBS Promela model of the simple

SDF graph with channel counters and a common buffer pool

(above) or separate buffers for each channel (below).

(safety properties), or to prove the existence of desired behaviour (live-

ness properties). As observed by GBS, when given an appropriate

Promela 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 7. Finally, as GBS

also used SPIN, the comparison between GBS and our approach is more

truthful.

4.1 GBS models with a common buffer pool

4.2 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 Se-

ction 3, and may therefore be considered a good concrete model. How-

ever, the GBS model for the case where instead of one buffer pool, each

channel has its own buffer space the model is not sufficiently abstract.

Figure 3 (below) presents the essence of this GBS model. 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 bound on two separate buffers is 6 (feasible property), and that no

periodic schedules are possible with a sum less than or equal to 5 (infe-

sible property).

In spite of the clever lower bound optimisation, the GBS model of

Figure 3 (below) has two problems. Firstly, the state space of this model

is potentially 2^{16} times as large as the state space of Figure 3 (above).

This is caused by adding the two byte variables s0 and s1. Secondly,

the state vector itself is larger by 2 bytes, and the product of the size

of the state space and the state vector determines the amount of mem-

ory needed in the search. To develop a better model, a more abstract

approach is needed.

4.3 Abstract models with node counters

Figure 4 presents the essence of such a more abstract model. There are

two essential differences with the GBS model.

Firstly, the GBS channel counters contain redundancy that can be

avoided by using node counters instead. It is easy to calculate the value

of the channel counters from the node counters (as shown by the macro

definitions for c0 and c1) but it is not possible the other way round.

The first advantage is that in most non-trivial SDF graphs there are

fewer nodes than edges (there could be O(N^2) different edges versus

N nodes), thus potentially reducing the size of the state space signif-

icantly. The second advantage is that optimisations are more effective

on node counters than on channel counters, as we shall see in the next

section.

Secondly, the GBS models produce too many schedules because sched-

ules such as a(a(babaac))∗ with a transient aa and a periodic part

(....)∗ are redundant. To avoid a schedule with a transient we use a

more abstract LTL property that ensures that the schedule begins and

ends in the initial state. This more abstract 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
Figure 4: The essence of our Promela model of the simple SDF graph with node counters and a common buffer pool (middle) or separate buffers for each channel (below). The top part is common to both models.

specify a sequence of states for which the property \( p \) 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 buffers the infeasible property of Figure 4 (middle) can be used. For this particular benchmark, as we argued in Section 3, 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 6 for more information about the benchmarks).

The model for the separate buffer pool (Figure 4 below) is the same as for the common buffer pool. The LTL property needed to check for infeasible schedules consists of as many conjunctions as there are channels, with each conjunct reducing the buffer space for its channel by 1. Our infeasible model has a conjunct per channel, whereas GBS have an extra state variable per channel; one could argue that this is equally bad. However, our feasible model has neither, and is thus potentially more efficient than that of GBS.

We show that the abstraction leading to the LTL formulae of Figure 4 is justified in two steps. Firstly, we refer to the work of Lee and Messerschmitt [5]. They define a class-S algorithm essentially as a simulation of the semantics of Section 3 and prove that a class-S algorithm finds a schedule with a minimum period (equal to the sum of all the elements of the repetition vector) if such a schedule exists. The crucial property of a class-S algorithm is that it fires each node at most as often as specified by the repetition vector. Secondly, we use the model checker twice: to establish that (1) with the feasible property a schedule is found, and (2) with the infeasible property no schedule is found. Combined we have that our approach is guaranteed to find a schedule with the minimum period and the minimum buffer size if one exists. In Section 7, however, we will use Branch and Bound techniques to find the optimal buffer size with a single run of SPIN. Then the LTL formulæ will not longer be required as the liveness property will be converted to a reachability property.

In the next section we will explore a number of optimisations that make use of the node counters.

5 Optimisations

Optimisations are indispensable to 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. Such optimisations take the form of abstractions that generate fewer states than the concrete model. An abstraction that may miss periodic schedules, whilst all schedules found are indeed correct is incomplete. An abstraction that that may yield incorrect schedules is unsound. All types of abstraction may be useful. For example a schedule found by an incomplete abstraction may be 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 abstraction is correct or not. We give a number of examples of effective optimisations, indicating whether the optimisation is sound and/or complete on a benchmark of commonly used SDF graphs.

5.1 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 \( nx \times y \). Sound, because we are not changing any schedules. Complete, because we implement in essence a class-S algorithm.

5.2 Clustering

In realistic data flow graphs the firing rate of some nodes may differ considerably. Figure 5 (above) gives an example of a chain from the h263 benchmark, 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 5 (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 \( \Gamma' \) and \( \Gamma'' \) be the topology matrix before and after the transformation. It is easy to check that \( rank(\Gamma') = rank(\Gamma'') \) hence the transformation does not affect consistency. However, the transformation is unsound, since \( lwb(c7) = k \) whereas \( lwb(c7) = 1 \).

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 \( (pqqr's)^* \) 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 \( (pq^kr's)^* \) for the original system. Unfortunately, this schedule requires a buffer of size \( k \) for channel \( c7 \).
shows for each benchmark the best results that we obtain in (above). Using we can also prove that this is an optimal schedule. Unsound, has been constructed such that there are two

5.3 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 6. 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 \( e_0 \) and \( c_2 \).

The example of Figure 6 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 \( p_4 \) of Figure 6 forces the excess to be stored in \( c_4 \), and property \( p_2 \) stores the excess in \( c_2 \). However, only one of these methods (i.e. property \( p_2 \)) is compatible with the optimisation for look ahead. To illustrate this point Figure 6 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 \( p_4 \) only allows the excess to be stored in \( c_4 \). If on the other hand we would have used property \( p_2 \), the network would not have been blocked.

This shows that look ahead is not a sound optimisation. In the adbetter benchmark, which has been carefully constructed by Adé to demonstrate the intricacies of SDF scheduling [2], look ahead will even increase the minimum buffer capacity by one, which makes it even less satisfactory as an optimisation. However, we have found look ahead to be effective in all benchmarks except the artificial examples! Sound (because we are only blocking nodes, not changing any parameters), Incomplete.

6 Checking the optimal buffer size

Our benchmark consists of 10 SDF graphs taken from various sources. The benchmarks simple, bipart, eddat, modem, ade, adebetter, inmarsat, and h263 are used by GBS [3], the benchmarks mp3sys and mp3dec are used by Stuik et al [8]. These benchmarks are used by many other authors in the field and are therefore assumed to be representative for SDF graphs.

To avoid the tedium of 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 165 different SPIN models to explore the optimisations described in Section 5 (and several other optimisations that were found to be ineffective, but which for lack of space cannot be described here), as well as all the combinations of optimisations that make sense.

Table 1 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. 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 (See below).

The rows marked GBS in the first column are the best results of GBS taken from their paper [3, 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 instead of channel counters. As explained before, this is due to the potential reduction of the number of state variables from quadratic in the number of nodes for channel counters) to linear in the number of nodes (for node 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 5.2 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 [9] and Hebe [10], 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 1. 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 considerably more time experimenting with SPIN.

### 7 Finding the optimal buffer size

Thus far we have provided improvements on 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 1 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.

#### 7.1 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. $\ellwb_{\ell}(\cdot)$ and $\upb_{\ell}(\cdot)$ in Section 3). 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 node $n$ 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 $a$, $b$, and $c$, 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 $c$ 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:

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 $\tilde{r}(n)$ times.

The lower bound on the total common buffer size $\ellwb_{\ell}(n)$ 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 $\ellwb_{\ell}(a) = 4$. The maximum is reached after two firings of $a$ as shown in Figure 7(a). Figure 7(b,c) show that $\ellwb_{\ell}(b) = 4$, and $\ellwb_{\ell}(c) = 2$. 

---

<table>
<thead>
<tr>
<th>Simple</th>
<th>Bipart</th>
<th>Cddat</th>
<th>Modem</th>
<th>Ade</th>
<th>Adebetter</th>
<th>Inmarsat</th>
<th>h263</th>
<th>mp3dec</th>
<th>mp3sys</th>
<th>5.2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GBS</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5.2</td>
</tr>
<tr>
<td>11</td>
<td>88</td>
<td>4127</td>
<td>210</td>
<td>497</td>
<td>8602†</td>
<td>2862</td>
<td>66</td>
<td>4758</td>
<td>9</td>
<td>139</td>
</tr>
<tr>
<td>5.1, 5.3</td>
<td>8</td>
<td>84</td>
<td>614</td>
<td>50</td>
<td>47</td>
<td>129*</td>
<td>1133</td>
<td>52</td>
<td>4758</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>16385</td>
<td>125</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**States stored checking feasible schedule with given bound. Separate buffer space**

<table>
<thead>
<tr>
<th>GBS</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2241</td>
<td>1708†</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>581</td>
<td>721</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**States stored checking feasible schedule with given bound. Separate buffer space**

<table>
<thead>
<tr>
<th>GBS</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>124</td>
<td>755</td>
<td>1231</td>
<td>366</td>
<td>180369</td>
<td>163</td>
<td>9511</td>
<td>11</td>
<td>23</td>
<td>55004</td>
</tr>
<tr>
<td>5.1, 5.3</td>
<td>8</td>
<td>94</td>
<td>614</td>
<td>176</td>
<td>96</td>
<td>109</td>
<td>1110</td>
<td>52</td>
<td>4758</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>16381</td>
<td>121</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**States stored checking feasible schedule with given bound. Common buffer space**

<table>
<thead>
<tr>
<th>GBS</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>150</td>
<td>3542</td>
<td>853</td>
<td>303</td>
<td>140</td>
<td>aborted</td>
<td>240</td>
<td>aborted</td>
<td>7</td>
<td>12</td>
</tr>
<tr>
<td>5.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>149</td>
<td>3541</td>
<td>852</td>
<td>127</td>
<td>139</td>
<td>13102300</td>
<td>81</td>
<td>2826250</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| **millisecond 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  |
| Hebe  | 11±3   | 11±3  | 11±3  | 15±3 | 12±3      | 12±3     | 16±3 | 15±3   | 12±4   | 14±2 |
| SPIN  | 18±8   | 20±8  | 20±9  | 19±8 | 41±16     | 41±19    | 24±9 | 18±9   | 33±10  | 18±9 |
|       | 70±12  | 19±8  |       |       |           |          |      |        |        |     |

Table 1: 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 [3, table 1])**
5.2 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 [7], 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: \( g \leftarrow \max_{1 \leq n \leq N} \text{lwb}_{n}(n) \), and \( s \leftarrow \min_{1 \leq n \leq N} \text{lwb}_{n}(n) \).

2. repeat
   2.1 Use SPIN to 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 [7] 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 SPIN with \( g + 1 \) (see Table 3 and the appendix), 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 3 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 once more, with an updated guess of \( g = 22 \). 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

<table>
<thead>
<tr>
<th>Initial guess ( g )</th>
<th>States stored</th>
<th>Feasible bounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>30</td>
<td>none</td>
</tr>
<tr>
<td>14</td>
<td>66</td>
<td>none</td>
</tr>
<tr>
<td>18</td>
<td>157</td>
<td>none</td>
</tr>
<tr>
<td>22</td>
<td>7648</td>
<td>21,20,19,18</td>
</tr>
<tr>
<td>total</td>
<td>7901</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>1568</td>
<td>18</td>
</tr>
<tr>
<td>ratio</td>
<td>5.0</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Branch and Bound strategy to search for the optimal common buffer size for adebetter. Step size \( s = 4 \).

To prove that \( \text{lwb}_{n}(n) \) 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 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 \( \text{lwb}_{n}(n) \) gives a lower bound on the amount of common buffer space required by node \( n \).

The complexity of the algorithm to calculate \( \text{lwb}_{n}(n) \) is linear in \( \bar{r}(n) \).

7.2 Optimisations for the minimum bound

--

Table 2: Buffer sizes and states stored ratios for the benchmarks. The top half applies to the common buffer case, the bottom half to the separate buffer case. \( \text{min}_{c,n} \) is the minimum buffer size required by a feasible schedule.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>common bipart</th>
<th>cdad</th>
<th>modem</th>
<th>ade</th>
<th>adebetter</th>
<th>inmars5.2</th>
<th>h2635.2</th>
<th>mp3dec</th>
<th>mp3sys5.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>s = ( \min_{1 \leq n \leq N} \text{lwb}_{n}(n) )</td>
<td>2</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>240</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>g = ( \max_{1 \leq n \leq N} \text{lwb}_{n}(n) )</td>
<td>4</td>
<td>16</td>
<td>15</td>
<td>10</td>
<td>25</td>
<td>9</td>
<td>720</td>
<td>4752</td>
<td>2</td>
</tr>
<tr>
<td>( \text{min}_{n} )</td>
<td>4</td>
<td>26</td>
<td>16</td>
<td>13</td>
<td>67</td>
<td>18</td>
<td>1008</td>
<td>4754</td>
<td>2</td>
</tr>
<tr>
<td>( \text{upb}<em>{n} = \sum</em>{1 \leq n \leq N} \text{lwb}_{n}(n) )</td>
<td>12</td>
<td>60</td>
<td>60</td>
<td>149</td>
<td>105</td>
<td>72</td>
<td>5472</td>
<td>23762</td>
<td>22</td>
</tr>
<tr>
<td>state stored ratio</td>
<td>1.0</td>
<td>1.0</td>
<td>1.9</td>
<td>1.5</td>
<td>3.9</td>
<td>5.0</td>
<td>3.9</td>
<td>1.4</td>
<td>1.0</td>
</tr>
<tr>
<td>SPIN runs</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>10</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>separate bipart</th>
<th>cdad</th>
<th>modem</th>
<th>ade</th>
<th>adebetter</th>
<th>inmars5.2</th>
<th>h2635.2</th>
<th>mp3dec</th>
<th>mp3sys5.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sum_{1 \leq n \leq C} \text{lwb}_{n}(n) )</td>
<td>6</td>
<td>28</td>
<td>32</td>
<td>38</td>
<td>49</td>
<td>39</td>
<td>3072</td>
<td>9508</td>
<td>12</td>
</tr>
<tr>
<td>( \text{min}_{c} )</td>
<td>6</td>
<td>28</td>
<td>32</td>
<td>38</td>
<td>83</td>
<td>42</td>
<td>3072</td>
<td>9508</td>
<td>12</td>
</tr>
<tr>
<td>( \sum_{1 \leq n \leq C} \text{upb}_{n}(n) )</td>
<td>8</td>
<td>264</td>
<td>1021</td>
<td>61</td>
<td>209</td>
<td>133</td>
<td>3936</td>
<td>9508</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 3: Buffer sizes and states stored ratios for the benchmarks. The top half applies to the common buffer case, the bottom half to the separate buffer case. \( \text{min}_{c,n} \) is the minimum buffer size required by a feasible schedule.

Figure 7: Common buffer space analysis for all three nodes of the SDF graph from Figure 1.
reasonable bound, as we can see by comparing the second row (labelled \( g = \max_{1 \leq x \leq N} lwb_n(x) \)) and the third row (labelled \( \min_{u_n} \)) that shows the true minimum bound in table 2 for all benchmarks. For completeness the table also shows the step size \( s = \min_{1 \leq x \leq N} lwb_n(x) \) and a (poor) upper bound calculated as \( \sum_{1 \leq x \leq N} lwb_n(x) \).

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 initial high 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 3 shows that 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 2 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.

8 Conclusions and future work

Many authors have used model checkers to solve scheduling problems, but Geilen, Basten and Stuijk [3] (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 lowers the search complexity of all benchmarks significantly in two ways: (1) the number of node firings is limited by the repetition vector (as opposed to not explicitly bounded for GBS), (2) the number of state variables is linear in the number of nodes (as opposed to quadratic for GBS), and (3) 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. 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. 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.

Acknowledgements

Maarten Wiggers ran our models through his Hebe tool. Marc Geilen and Sander Stuijk made the GBS benchmarks and the SDF3 tools available and gave helpful feedback on the paper. Gerard Holzmann answered all our SPIN questions. Maarten Wiggers, Angelika Mader, and Hylke van Dijk provided helpful feedback on the work.

References

Appendix – can be omitted from the proceedings

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
int __best = 5
```

```
#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{
    :: 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;
    }
}
```

```
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 ${promela_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 -DSAFeT ppan.c
```

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

For a full explanation of the mechanisms used please consult Ruys [7].