Minimising Buffer Requirements of Synchronous Dataflow Graphs with Model Checking

Marc Geilen, Twan Basten and Sander Stuijk
Eindhoven University of Technology, Department of Electrical Engineering
{m.c.w.geilen,a.a.basten,s.stuijk}@tue.nl

ABSTRACT

Signal processing and multimedia applications are often implemented on resource constrained embedded systems. It is therefore important to find implementations that use as little resources as possible. These applications are frequently specified as synchronous dataflow graphs. Communication between actors of these graphs requires storage capacity. In this paper, we present an exact method to determine the minimum storage capacity required to execute the graph using model-checking techniques. This can be done for different measures of storage capacity. The problem is known to be NP-complete and because of this, existing buffer minimisation techniques are heuristics and hence not exact. Modern model-checking tools are quite efficient and they have been successfully applied to scheduling-related problems. We study the feasibility of this approach with examples.

Categories and Subject Descriptors: C.3 [Special-purpose and Application-based Systems]: Signal processing systems

General Terms: Algorithms, Experimentation, Theory, Verification.

Keywords: Synchronous Dataflow, buffering, model-checking, optimization.

1. INTRODUCTION

1.1 Problem Statement

Synchronous Dataflow Graphs (SDFs) [10] are representations of signal processing applications or multimedia applications that allow for powerful analysis and synthesis techniques. The techniques can be used to quickly and automatically generate efficient implementations of signal processing applications. Since these implementations are typically used for embedded systems or systems on chip, resource usage (computation, energy, and so forth) is of eminent importance. Memory requirements should be kept as small as possible. Part of the memory requirements are in the data that needs to be stored in the channels connecting the computational components. Traditionally, SDFs have been mainly used for synthesis of sequential DSP programs in which case both code and data memory are important [5, 10, 12]. SDFs are also being used to analyse or design multiprocessor systems on a chip, possibly based on network-on-chip communication infrastructures [14]. The aim of such systems is to realise a predictable performance. In this case, the length of the schedule does not contribute to the code size, the elements of the graph are mapped on a parallel architecture. Scheduling is on-line and data driven. Minimising buffer capacity however is very important to reduce cost and improve energy efficiency. In this paper, we study the problem of minimising buffer storage capacity by using techniques and tools from the domain of model-checking.

Some examples of SDFs are depicted in Figures 1, 4, 5 and 6 (taken from [5]), 7 (from [1]) and 8. The nodes of these graphs are called *actors*; they represent functions that are computed by reading data items from their input ports, and writing the results of the computation as tokens on the output ports. An essential property of synchronous dataflow graphs is that every time an actor *fires* (performs a computation), it consumes the same amount of tokens from its input ports and produces the same amount of tokens on its output ports. These amounts are called the *rates* of the ports. The edges in these graphs represent data that is communicated from one actor to another. To allow actors to execute asynchronously, some storage capacity is required for these channels to store data that is produced by one actor, but has not yet been consumed by the next.

If the storage capacity in the channels that connect output ports to input ports is insufficient, an actor may not be able to fire, which in turn may lead to a deadlock situation in which the graph can no longer perform its computations. The buffer minimisation problem for SDF graphs as we consider it in this paper is to find the smallest size storage capacity for which the graph can be executed without the risk of a deadlock and to determine a schedule that realises a continuing, infinite execution within these bounds. Note that a further increase in storage capacity could increase the throughput of the dataflow graph [8], but studying this trade-off is outside of the scope of this paper. The schedule itself is not always explicitly required; a data driven execution of the graph with the computed storage capacities realises the correct schedule automatically. Variations of the specific measure of required storage capacity can be used [12]. One option is to look at every instant of the schedule and determine the amount of memory that is being used by all the channels together. For the whole schedule, the

DAC 2005, June 13–17, 2005, Anaheim, California, USA.

maximum of the sum of used storage is required. This is appropriate if memory can be shared between buffers. The life times of data items in the memory can be taken into account (this approach is taken in [12, 13]). Another variation is to consider the buffers between actors as being mapped onto separate storage elements, so empty space in one cannot be used for the other. Then, the maximum amounts over the entire schedule need to be determined per buffer, and the total amount of memory required is obtained by adding them up. This approach is proposed in [1, 5]. Our approach allows modelling of different measures of storage, including both mentioned above as well as hybrid forms of these measures.

The buffer minimisation problem is known to be NP-complete [4]. However, the graphs that we want to analyse are often moderately sized. Advances in model-checking techniques have improved their efficiency and they have been successfully applied to solve different NP-complete (and even worse) scheduling problems [2, 3, 16]. In [3], for instance, the Times tool uses a timed-automata based model-checker for solving real-time scheduling problems. [16] uses model-checking to determine low power schedules. These applications have inspired us to try to apply them to the buffer minimisation problem as well.

The contribution of this paper is a technique for reducing the SDF buffer minimisation problem to a model-checking problem on the state-space of the graph. It is shown to be amenable to different measures of buffer storage capacity and provides an exact solution to the minimisation problem. The principle is proved on a number of SDF graphs.

1.2 Related Work

Buffer minimisation. Minimisation of buffer requirements has been studied before by several authors. The proof of NP-completeness is given in [4]. Traditionally, for DSP software synthesis, a so-called single appearance schedule (SAS) is used to minimise code size. In a SAS, the functional code of the actors is included in a nested loop structure, such that each piece of code occurs only once. Then, amongst the different ways of defining the nested loop structure, the one with the least buffer requirements is chosen. Heuristics for finding such SAS schedules can be found in [5]. The SAS constraint is somewhat relaxed in [17], allowing for a reduction in buffer memory size. In the standard approach, it is assumed that every buffer has its own private part of the memory. Exploiting the life times of data elements, it can be advantageous to share the same piece of memory between buffers [12, 13, 16].

In a parallel architecture, where the length of the schedule does not lead to extra code size, further reductions are possible. This is for instance the case if we use SDFs for modelling networks-on-chip [14], parallel implementations on FPGA [1] or use a data-driven form of scheduling such as static order scheduling. This context is what this paper is concerned with and this approach is also taken in [1, 11]. [1] and [11] present different heuristic algorithms to determine small but sufficient buffer sizes. [7] presents an approach for buffer minimisation for on-line EDF scheduled dataflow graphs and demonstrates that on-line scheduling can cause large memory gains over off-line scheduling techniques. The method works on acyclic graphs and does not necessarily give optimal results. In contrast, this paper investigates an exact method to solve the same problem and investigates its feasibility in the light of the problem's complexity. Moreover, our method can deal with different measures of buffer capacity. [8] focusses on buffer minimisation for rate-optimal schedules. Only those schedules that achieve the maximal throughput are considered as potential candidates. The obtained buffer sizes are not always the smallest possible. The problem is solved by constructing an (integer) linear programming formulation of the problem and using linear programming algorithms for efficiency.

[18] presents a minimisation method giving close to minimum results for separate memories per buffer. The complexity of the algorithm is exponential and it yields exact minimum buffer requirements only for a subclass of networks.

Model-checking Techniques for Scheduling Problems. Besides function hardware or protocol verification (such as [15]), model-checking has been used in the past for many kinds of scheduling and scheduling-related problems [2, 3, 16]. It has often been shown that it can be competitive compared to existing scheduling heuristics and allows for heuristics, and suboptimal results as well, in case the problems are too large to be dealt with in an exact manner.

1.3 Paper Overview

The remaining parts of the paper are structured as follows. Section 2 introduces an operational semantics of SDF. Section 3 defines scheduling of the SDF graphs and computation of the corresponding buffer requirements. The reduction of the buffer minimisation problem to a model-checking problem is discussed in Section 4. Section 5 deals with a generic measure of storage capacity. Experimental support for the approach can be found in Section 6. Section 7 discusses complexity issues of the problem and the approach of this paper and Section 8 concludes.

2. OPERATIONAL SEMANTICS OF SDF

We first formalise the behaviour of an SDF graph using a transition system. Since we are only interested in the *amount* of data stored in channels, we abstract from the actual data that is being communicated and data transformations that are being performed by the actors. A similar approach, which does include data, is suggested in [6]. We initially assume that all channels have unbounded capacity.

Let $\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$ be the set of natural numbers, extended with infinity (∞). To measure quantities related to channels, such as the number of tokens present in, read from or written to channels, or the capacities of channels, we define the following structure.

Definition 1. (Channel Quantity) *A channel quantity on a set C of channels is a mapping $C \to \mathbb{N}^\infty$. If δ_1 is a channel quantity on C_1, δ_2 on C_2, $C_1 \subseteq C_2$ and for every $c \in C_1$, $\delta_1(c) \leq \delta_2(c)$, we write $\delta_1 \preceq \delta_2$.*

The set of channel quantities with the relation $\preceq$ forms a bounded complete partial order (cpo). A channel quantity δ is finite if for every channel c, $\delta(c) < \infty$ (i.e. if it is a finite element of the cpo). If δ_1 and δ_2 are finite channel quantities on C and $\delta_1 \preceq \delta_2$, then $\delta_2 - \delta_1$ denotes the channel quantity δ such that $\delta(c) = \delta_2(c) - \delta_1(c)$ for all $c \in C$. Similarly, $\delta_1 + \delta_2$ denotes the channel quantity obtained by adding the individual quantities, $(\delta_1 + \delta_2)(c) = \delta_1(c) + \delta_2(c)$.

Definition 2. (Port) *We assume a set Ports of ports and with every $p \in Ports$, we associate a rate $Rate(p) \in \mathbb{N}$.*

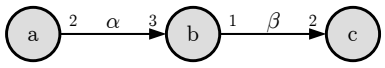

Figure 1: Example SDF graph

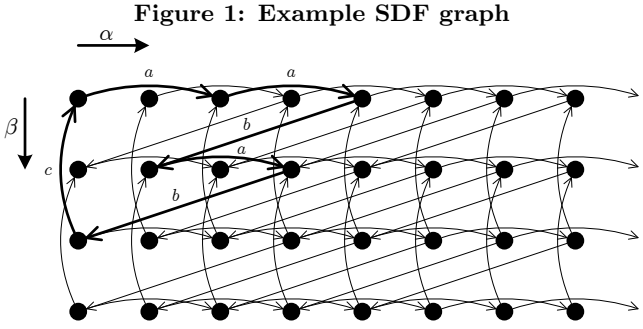

Figure 2: State space of the example SDF graph

Definition 3. (Actor) *An actor $a = (In, Out)$ consists of a set $In \subseteq Ports$ (denoted by $In(a)$) of input ports and a set $Out \subseteq Ports$ (denoted by $Out(a)$) of output ports.*

The execution of an actor is defined by discrete *firings*. When an actor fires, it consumes $Rate(p) \in \mathbb{N}$ tokens on every input port $p \in In(a)$ and produces $Rate(p) \in \mathbb{N}$ tokens on every output port $p \in Out(a)$. A firing can be described by a channel quantity $Rd(a) = \{(p, Rate(p)) \mid p \in In(a)\}$ describing the tokens that are read and $Wr(a) = \{(p, Rate(p)) \mid p \in Out(a)\}$ for the tokens that are written.

Definition 4. (Synchronous Dataflow Graph) *An SDF graph is a tuple (A, C) consisting of a set A of actors and a set C of channels. Every channel $c \in C$ is connected to its source, one output port $Src(c)$ of an actor $a \in A$, and to its destination, one input port $Dst(c)$ of an actor $a \in A$. Every port is connected to at most one channel.*

An SDF graph may have input and output ports in the form of ports of its actors that are not connected to channels.

Definition 5. (Configuration) *A* configuration *of an SDF graph (A, C) is a channel quantity γ on C, that associates with every channel $c \in C$, the number of tokens present in that channel in that configuration.*

Definition 6. (Firing) *Given a configuration γ of SDF graph (A, C). Actor $a \in A$ can fire if $Rd(a) \preceq \gamma$, resulting in the configuration $\gamma - Rd(a) + Wr(a)$. This is denoted as $\gamma \rightarrow \gamma - Rd(a) + Wr(a)$.*

Definition 7. (SDF State Space) *Firings define a relation on graph configurations. With an SDF graph (A, C), we associate the state space $(\Gamma, \rightarrow)$, where Γ is the set of configurations of C.*

Example. In Figure 1, the SDF graph $(\{a, b, c\}, \{\alpha, \beta\})$ (consisting of actors a, b and c and channels α and β) is shown. $a = (\emptyset, \{p\})$, $b = (\{q\}, \{r\})$ and $c = (\{s\}, \emptyset)$. $Src(\alpha) = p$, $Rate(p) = 2$, $Dst(\alpha) = q$, $Rate(q) = 3$, $Src(\beta) = r$, $Rate(r) = 1$, $Dst(\beta) = s$, $Rate(s) = 2$. From the initial configuration $\{(\alpha, 0), (\beta, 0)\}$ with empty buffers, a firing of actor a changes the configuration to $\{(\alpha, 2), (\beta, 0)\}$ having 2 tokens on channel α. Actor b cannot fire yet, since it requires 3 input tokens. A second firing of actor a changes the configuration to $\{(\alpha, 4), (\beta, 0)\}$. Now b is enabled and when it fires, the configuration becomes $\{(\alpha, 1), (\beta, 1)\}$. Similarly, as illustrated by the bold transitions in Figure 2 that shows the state space of the graph, subsequent firings of actors a, b and c return the graph to the initial configuration.

3. SCHEDULING

Every path in the state space of an SDF graph constitutes a schedule of that graph.

Definition 8. (Schedule) *Let (A, C) be an SDF graph. A schedule of (A, C) is a finite or infinite sequence $\sigma = \gamma_0\gamma_1\gamma_2 \ldots$ of channel quantities on C such that for all $0 \leq i < |\sigma|$, $\gamma_i \rightarrow \gamma_{i+1}$.*

Note that although the schedule contains only the channel configurations, from such a sequence, one can derive the corresponding sequence of actor firings. We assume that the initial token distribution is given. We use $\sigma_1 \cdot \sigma_2$ to denote the concatenation of schedules σ_1 and σ_2 and σ^∞ to denote the infinite (periodic) repetition of finite schedule σ. We also use $\sigma(i)$ to denote γ_i.

Definition 9. (Periodic Schedule) *An infinite schedule σ is* periodic *iff it is of the form $\sigma = \sigma_{pre} \cdot \sigma_{per}^\infty$, with a finite prefix σ_{pre} followed by a repetition of schedule σ_{per}.*

For software synthesis, a periodic schedule is sought, to be converted into an infinite loop implementing the graph. Given a schedule, we can determine the buffer storage that is required, being the least upper bound of the channel quantities along the schedule. This is the maximum amount of storage used, or ∞ if such a maximum does not exist.

Definition 10. (Channel Bounds) *Let σ be a schedule of SDF graph (A, C). The required channel bounds, are given by $\bigsqcup\{\sigma(i) \mid 0 \leq i < |\sigma|\}$.*

Definition 11. (Storage Measure) *Let σ be a schedule of (A, C). Storage measure type (a), the total size of separate memories per channel, for schedule σ, equals $\sum_{c \in C} (\bigsqcup\{\sigma(i) \mid 0 \leq i < |\sigma|\})(c)$ and type (b), the size of a shared memory for all channels, as $\bigsqcup\{\sum_{c \in C} \sigma(i)(c) \mid 0 \leq i < |\sigma|\}$.*

($\bigsqcup X$ denotes the least upper bound of the set X w.r.t. the corresponding order.) In the remainder of Section 3 and Section 4, we focus on type (a). In Section 5, we introduce a generalised storage measure that covers both (a) and (b).

Theorem 1. *There exists a schedule σ for SDF graph (A, C), with finite channel bounds β, iff there exists a periodic schedule with channel bounds β.*

Proof. Let σ be a non periodic schedule with channel bounds β. The number of configurations within bounds β is finite, thus there is a configuration γ that appears infinitely often in the schedule. Thus, the schedule has the form $\sigma_{pre} \cdot \gamma \cdot \sigma_1 \cdot \gamma \cdot \sigma_2$. Because the configurations are memory-less, $\sigma_{pre} \cdot \gamma \cdot (\sigma_1 \cdot \gamma)^\infty$ is a periodic schedule within bounds β. □

According to Theorem 1, it is sufficient to test if the graph with bounds allows a periodic schedule to determine whether it allows any schedule at all. In the next section, we show how a model-checking tool can be used to determine if a periodic schedule exist for given channel bounds.

4. MODEL-CHECKING APPROACH

To apply the model-checking approach, we encode the operational semantics of SDF in the model-checker Spin [9]. Additionally, we encode the desired channel bound constraints. One typical approach to solve scheduling problems with a model-checking tool is to formulate the logical property that an execution satisfying all constraints does not exist ([2, 3]). Hence, we challenge Spin to disprove our claim

that an execution within given channel bounds does not exist. If SPIN proves us wrong, it produces a counter example. This is an execution (a schedule) of the system that satisfies all constraints. The counter examples it finds are always periodic. From Theorem 1, we know that for bounded-channel constraints, if such a schedule does not exist, then no schedule exists. Iteratively, we can now vary the memory limit, applying a binary search (hence, few iterations will be required) for the lowest amount that allows a schedule to be constructed. One can test whether such a bound exists using the balance equations [10]. An upper bound on the channel bounds can be derived from the balance equations, a tighter one from an existing sub-optimal buffer minimisation method. This makes that the iterative approach described above always ends. (At least in principle. In practice, memory and time requirements of the algorithm may prevent this.) We continue to make the approach sketched above precise. To this end, we now define the actual state-space that will be computed by the model-checker, the state space of the SDG graph extended with information on the channel bounds required up to that point.

Let $(\Gamma, \rightarrow)$ be the state space of the SDF graph. Define the *model-checking state space* $(\Gamma \times \Gamma, \Rightarrow)$ as follows. The states are elements of $\Gamma \times \Gamma$; the first channel quantity is the current configuration and the second encodes the storage bounds required for the schedule so far. Then $(\gamma, b) \Rightarrow (\gamma', b')$ iff $\gamma \rightarrow \gamma'$ and $b' = b \sqcup \gamma'$, b' is the least upper bound of b and the new configuration γ and hence sufficiently large for the schedule including the new configuration γ'.

It is easy to see that in the model-checking state space, the second channel quantity computes the required channel bounds. For a finite schedule, the bounds in the last configuration are the required channel bounds; for infinite schedules, the bounds converge to the required bounds, or in technical terms, they form a chain of bounds with as least upper bound the required channel bounds.

LEMMA 1. *Let $\sigma = \gamma_0\gamma_1 \ldots$ be a schedule of $(\Gamma, \rightarrow)$. There exists a path $\tau = (\gamma_0, b_0)(\gamma_1, b_1) \ldots$ of $(\Gamma \times \Gamma, \Rightarrow)$ with $b_0 = \gamma_0$ in the model-checking state space, such that $\{b_i \mid 0 \leq i < |\sigma|\}$ is a chain of channel bounds and $\bigsqcup_{0 \leq i < |\sigma|} b_i = \bigsqcup_{0 \leq i < |\sigma|} \gamma_i$. In particular, if σ is finite, then $b_{|\sigma|-1} = \bigsqcup_{0 \leq i < |\sigma|} \gamma_i$.*

The proof is straightforward. Note that from this lemma it follows that the model-checking state space computes the channel bounds explicitly.

THEOREM 2. *The state space $(\Gamma \times \Gamma, \Rightarrow)$ has a periodic schedule with length $n \in \mathbb{N}$ iff the state space $(\Gamma, \rightarrow)$ has a periodic schedule with length n.*

PROOF. A periodic schedule of $(\Gamma \times \Gamma, \Rightarrow)$ can be transformed into a periodic schedule of $(\Gamma, \rightarrow)$ by removing the additional information on the bounds. We continue with the proof of the other direction. Let $\sigma = \sigma_{pre}\sigma_{per}^{\infty}$ be a periodic schedule of $(\Gamma, \rightarrow)$. We 'simulate' the prefix $\sigma_{pre}\sigma_{per}$ on $(\Gamma \times \Gamma, \Rightarrow)$ starting from $(\sigma(0), \sigma(0))$ which gives the schedule τ_{pre} ending in (γ, b). Let $\sigma_{per} = \gamma_0\gamma_1 \ldots \gamma_{n-1}$. $\gamma_{n-1} \rightarrow \gamma_0$ and hence $(\gamma_{n-1}, b) \Rightarrow (\gamma_0, b)$ since $\gamma_i \preceq b$ for all $0 \leq i < n$ by Lemma 1. Similarly $(\gamma_0, b) \Rightarrow (\gamma_1, b)$ and so forth for σ_{per} with bound b leading to $(\gamma_{n-1}, b) = (\gamma, b)$ completing the cycle τ_{per} of length n. Hence, $\tau_{pre}\tau_{per}^{\infty}$ is a periodic schedule of $(\Gamma \times \Gamma, \Rightarrow)$ with length n. □

We now show how the model-checking state space can be encoded in PROMELA, the modelling language of SPIN. The

```
#define UPDATE(c) if :: ch[c]>sz[c] -> sz[c] = ch[c] :: else fi
#define PRODUCE(c,n) ch[c] = ch[c] + n; UPDATE(c)
#define CONSUME(c,n) ch[c] = ch[c] - n
#define WAIT(c,n) ch[c]>=n

int ch[2]; int sz[2];

proctype Actor_a(){
    do
    ::   atomic{
         PRODUCE(0,2);}
    od}
proctype Actor_b(){
    do
    ::   atomic{
         WAIT(0,3) ->
         CONSUME(0,3);
         PRODUCE(1,1)}
    od}
```

```
#define BOUND 6
#define SUM sz[0]+sz[1]
<> SUM>BOUND
```

```
proctype Actor_c(){
    do
    ::   atomic{
         WAIT(1,2) ->
         CONSUME(1,2);}
    od}
init{
    atomic{
    run Actor_a();
    run Actor_b();
    run Actor_c();}}
```

Figure 3: Promela code of an SDF graph

model corresponding to the example of Figure 1 is shown in Figure 3. Every SDF graph model uses a global variable `ch`, an array of integers, to encode the number of tokens in the individual channels in the current configuration. The array has two elements representing the two channels α (`ch[0]`) and β (`ch[1]`). Another global array `sz` is used to remember the maximum number of tokens that were stored in a channel at any given moment of the schedule. Thus, it encodes the bound $b \in \Gamma$ of a state (γ, b) in the model-checking state space. The `#define` directives at the top facilitate the modelling of actors by translating synchronisation, production and consumption of tokens into operations on `ch` and `sz`. `PRODUCE(c,n)` (produce n tokens on channel c) for instance is implemented as adding n to the appropriate element of the channel array and after that update the bounds if necessary (`UPDATE(c)`). `WAIT(c,n)` is used as a condition; the execution of the actor is blocked until channel c contains sufficient (n) token for the immediately following `CONSUME` operation. The `atomic{...}` clauses ensure that the statements inside are executed together as one, single transition. The `proctype` definitions define the behaviour of the individual actors. `do :: ... od` specifies an infinite loop. Inside the loop are the `WAIT`, `CONSUME` and `PRODUCE` operations appropriate for the actor. Finally, the `init` clause tells SPIN that of every type of actor, one instance should be run.

The verification challenge can be formulated as: "Every schedule will eventually require a storage capacity larger than *bound*". The PROMELA specification of the corresponding formula (for type (a)) in Linear Temporal Logic is shown in the box in Figure 3. It states that eventually (`<>`) the sum of the memories needed by the individual channels (`SUM=sz[0]+sz[1]`) will exceed the imposed bound `BOUND=6`. If this claim is false, SPIN will provide a counter example, which is a schedule within the required storage bounds.

5. GENERIC MEASURE OF STORAGE

We have seen that depending on the memory organisation of the system described by the SDF graph, different measures of storage may be appropriate. In this section, we generalise these measures to include both variations described earlier, as well as hybrid situations.

Our model of the memory organisation of the buffers is that the channels of the SDF graph can be partitioned into sets of channels that each use a shared memory for their storage. Across these partitions, channels cannot share data. In a network-on-chip architecture for instance, communication channels within a local tile can be mapped on a single mem-

ory, while channels across tiles reside in separate memories. Note that this model generalises both cases, (a) and (b), described earlier. The finest possible partitioning assumes a separate memory for every channel, while the coarsest partitioning uses a single memory for all channels.

Let $\mathcal{P}$ be a partitioning of the channels C of SDF graph (A, C). A channel bound is then a structure that contains the required memory size for every set of channels in $\mathcal{P}$: $\mathcal{P} \to \mathbb{N}^\infty$. The bound now is an upper bound on the memory used by the channels of one partition combined, at any given point in time. Given a configuration γ of the SDF graph, its memory requirements can be computed as follows: $\sum_{\mathcal{P}} \gamma$ which denotes $\{(\pi, s) \mid \pi \in \mathcal{P}, s = \sum_{c \in \pi} \gamma(c)\}$. The memory requirements are expressed in terms of token sizes. If the tokens in different channels have different sizes, an additional weight factor w_c (equal to the token size) can be introduced for every channel $c \in C$: $\{(\pi, s) \mid \pi \in \mathcal{P}, s = \sum_{c \in \pi} w_c \cdot \gamma(c)\}$. The total storage required for a schedule is again obtained by computing the least upper bound of the storages for the individual configurations and summing up the requirements per partition. It is not hard to verify that this generalised measure of storage capacity can be used with the method described in this paper, in essentially the same way as the simpler one used to explain the procedure.

6. EXPERIMENTS

We know that we cannot hope that our algorithm always quickly produces an answer to the problem. The state-space of a dataflow graph can be exponential in its size. We have performed a few experiments to see how the approach performs in practice, for the two types of storage requirements, separate memories per channel (a) and shared memory for all channels (b). From [1, 11], we know that a lower bound on the memory on an edge with production rate p, consumption rate c and initial number of tokens t can be computed as $p + c - \gcd(p, c) + t \bmod \gcd(p, c)$ (naturally, t itself is also a lower bound). This lower bound can be used to initialise the size to a value closer to the final channel sizes, which speeds up the model-checking analysis. We have done the experiments starting from 0 size channels as well as starting from these lower bounds. We have used the simple example of Figure 1, the example SDF graphs of [5], depicted in Figures 4, 5 and 6, a graph from [1] (Figure 7), a constructed graph for which [1] does not compute the optimal bounds (Figure 8), the INMARSAT mobile receiver of [7] (the main strongly connected component only, see Section 7) and a model of an H.263 decoder.

Table 1 shows the results for storage requirements of type (a), with (*low. bounds*) and without (*zero*) lower bounds, and type (b). The minimum total buffer capacity is shown, as well as the sizes of the state spaces examined by the model-checker to prove the feasibility of the given bounds (*St. space sched.*) and the one to demonstrate the infeasibility of smaller bounds, the minimum bounds minus one (*St. space infeas.*) respectively. We have also measured computation time of the results and the amount of memory that was used on a 1.5GHz Pentium IV PC. Computation time is directly related to the size of the state space and typically smaller than 1s in the small cases. Showing infeasibility for the sample rate conversion for instance took 3.8s and the modem 117.6s. Similarly, memory usage was very low, except for the same cases: 16Mb for infeasibility of the sample rate conversion and 419Mb for infeasibility of the

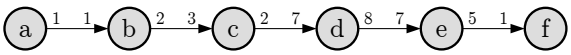

Figure 4: Sample rate conversion ([5])

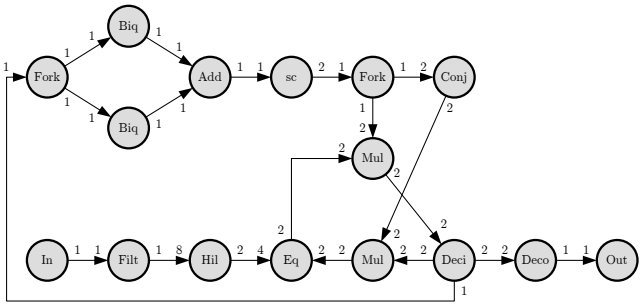

Figure 5: Modem application ([5])

modem, 2Gb was used for the cases that failed to produce an answer (marked with (*) in the table).

Note that the experiments with adding lower bounds show that for separate memories many examples do not require more memory than the lower bound! We also found that the feasibility proofs were often given very quickly, while infeasibility proofs failed to complete; a valid schedule can be found in many places in the state space. To prove absence of a schedule, the entire state-space needs to be searched.

The INMARSAT model has large up- and down-sampling coefficients (240 to 1). The same is true for the H.263 decode (2376 to 1, but for fewer channels). Interleaving the many individual read and write actions blows up the state space.

7. COMPLEXITY ISSUES

The buffer minimisation problem is NP-complete [4]. Efficient heuristic algorithms exist. Model-checking is linear in the size of the state space and PSPACE in the length of the formula to be checked. In our method, the length of the formula is constant, independent of the size of the graph. The size of the state space however, can be exponential in the size of the SDF graph.

We have explored the possibilities of using model-checking techniques for buffer minimisation, but we feel that there is still some room left for improvements in efficiency of the approach. Dataflow graphs are deterministic and the state space could be reduced by exploiting this determinacy using partial order reduction techniques. Moreover, it might be possible to apply some of the non-exact techniques or heuristics in the model-checking approach as well.

From [1, 11], we know a lower bound on the memory on an edge with production rate p, consumption rate c and initial number of tokens t. We have used this lower bound to initialise the size to a value closer to the final channel sizes, speeding up the model-checking analysis. These lower bounds are in fact sufficient for edges that are not on a cycle (when edges are interpreted as undirected edges, two parallel branches in the graph may constitute a cycle). This allows breaking up an SDF graph into its strongly connected components and perform the analysis on the components individually (as we did for the INMARSAT mobile receiver). Then, for the channels between the strongly connected components, the computed lower bounds suffice. From this, it follows directly, that for the graph of Figure 1, the lower bounds are sufficient. The same holds for the graph of Figure 4. An upper bound for the total amount of memory required for all channels can be derived from existing, heuristic buffer

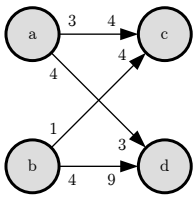

Figure 6: Bipartite graph ([5])

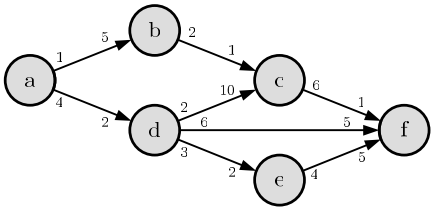

Figure 7: Example SDF graph ([1])

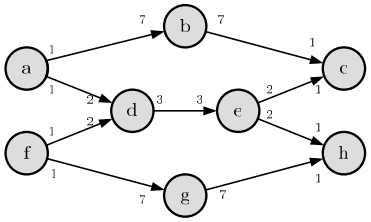

Figure 8: Example graph with optimum between bounds of [1]

Table 1: Experimental results

	Simple	*Bipartite*	*Samp.Rate*	*Modem*	*Ex. [1]*	*Fig. 8*	*Inmarsat [7]*	*H.263*
Stor. req. (a)	6	28	32	38	83	42	1508	7133
Lower bound	6	28	32	38	49	39	1508	7133
St. space sched., zero	13	129	4281	447	1204	242696	$> 11.2 \cdot 10^6$(*)	9514
St. space infeas., zero	9	48	193441	$2.03 \cdot 10^6$	7751	192099	$> 11.2 \cdot 10^6$ (*)	$> 22.3 \cdot 10^6$ (*)
St. space sched., low. bounds	11	88	4127	210	497	1708	2862	4758
St. space infeas., low. bounds	2	2	2	2	2441	8602	2	2
Stor. req. (b)	4	26	16	13	67	18	≤ 732	4754
State space sched.	9	124	755	1231	366	156	180369	9511
State space infeas.	4	150	3542	853	303	140	$> 17.9 \cdot 10^6$ (*)	$5.7 \cdot 10^6$

sizing algorithms such as [1]. In many of the SDF graphs we found, there is little or no space between these bounds and the minimal bounds are relatively easily proved.

8. CONCLUSIONS

In this paper, we have presented a method to obtain the exact minimum memory requirements for deadlock-free execution of an SDF graph. It differs from existing minimisation methods, which revert to heuristics to deal with the complexity of the problem. Different measures of storage are considered. Experiments on some well-known examples of SDF graphs show encouraging results, although further improvements are necessary. Future work includes further analysis of scalability of the method for larger graphs, how the approach can be further optimised by taking into account additional properties of the problem domain, such as the determinacy of dataflow models. Another important direction for future work is the extension of the model with timing information, so that the throughput vs. memory tradeoff can be explored. Since the behaviour of the SDF graph is modelled very operationally and straightforwardly in PROMELA, it is fairly easy to add additional constraints or implementation details. This way it might for instance be possible to encode the constraint to single appearance schedules. Additionally, the framework presented in this paper is in principle not restricted to SDF. It would be interesting to investigate if the approach could be made to work for instance for some BDF graphs. (That problem is undecidable in general, but that does not exclude that it might work for many BDF graphs in practice.)

Acknowledgement. This work is supported by the IST-2000-30026 project, Ozone.

9. REFERENCES

[1] M. Adé, R. Lauwereins, and J. A. Peperstraete. Data memory minimisation for synchronous data flow graphs emulated on dsp-fpga targets. In *Proc. 34th DAC, pages 64–69, 1997. ACM Press.*

[2] K. Altisen, G. Gößler, and J. Sifakis. A methodology for the construction of scheduled systems. In *Proc. FTRTFT 2000*, pages 106–120. Springer, 2000.

[3] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Times: a tool for schedulability analysis and code generation of real-time systems. In *Proc. of FORMATS'03*, number 2791 in LNCS, pages 60–72. Springer–Verlag, 2004.

[4] S.S. Bhattacharyya, P.K. Murthy, and E.A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer, 1996.

[5] S.S. Bhattacharyya, P.K. Murthy, E.A. Lee. Synthesis of embedded software from synchronous dataflow specifications. *VLSI Sig. Proc. Syst.*, 21(2):151–166, 1999.

[6] J.T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. PhD thesis, University of California, EECS Dept., Berkeley, CA, 1993.

[7] S. Goddard and K. Jeffray. Managing memory requirements in the synthesis of real-time systems from processing graphs. In *Proc. 4th IEEE Real-Time Technology and Applications Symp., June 1998*, pages 59–70, 1998.

[8] R. Govindarajan, G.R. Gao, and P. Desai. Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks. *J. of VLSI Sig. Proc.*, 31:207–229, 2002.

[9] G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison Wesley, 2004.

[10] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *IEEE Proceedings*, 75(9):1235–1245, September 1987.

[11] P. K. Murthy. *Scheduling Techniques for Synchronous and Multidimensional Synchronous Dataflow*. PhD thesis, University of California, Berkeley, December 1996.

[12] P. K. Murthy and S. S. Bhattacharyya. Shared memory implementations of synchronous dataflow specifications. In *Proc. DATE, March 2000. Paris, France*, pages 404–410. IEEE Computer Society Press, 2000.

[13] H. Oh and S. Ha. Efficient code synthesis from extended dataflow graphs for multimedia applications. In *Proc. 39th DAC, 2002, New Orleans, LA, USA*, pages 275–280. IEEE Computer Society, June 2002.

[14] P. Poplavko, T. Basten, M. Bekooij, J. van Meerbergen, and B. Mesman. Task-level timing models for guaranteed performance in multiprocessor networks-on-chip. In *Proc. of CASES 2003*, pages 63 – 72, IEEE CS Press , 2003.

[15] A. Roychoudhury, T. Mitra, and S. R. Karri. Formal verification of a system-on-chip bus protocol. In *Proc. of DATE 2003, Munich, Germany*, IEEE CS Press, 2003.

[16] S.K. Shukla and R.K. Gupta. A model checking approach to evaluating system level dynamic power management policies for embedded systems. In *Proc. HLDVT'01*, pages 53-57. IEEE Comp. Soc., 2001.

[17] W. Sung, J. Kim, and S. Ha. Memory efficient software synthesis from dataflow graph. In *Proc. ISSS '98*, pages 137–144. IEEE Computer Society, 1998.

[18] M. Čubrić and P. Panangaden. Minimal memory schedules for dataflow networks. In *Proc. CONCUR'93*, LNCS Vol. 715, Springer, 1993.