Modeling Static-Order Schedules in Synchronous Dataflow Graphs

Morteza Damavandpeyma[1], Sander Stuijk[1], Twan Basten[1,2], Marc Geilen[1] and Henk Corporaal[1]
[1]Department of Electrical Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands
[2]Embedded Systems Institute, Eindhoven, The Netherlands
{m.damavandpeyma, s.stuijk, a.a.basten, m.c.w.geilen, h.corporaal}@tue.nl

***Abstract*—Synchronous dataflow graphs (SDFGs) are used extensively to model streaming applications. An SDFG can be extended with scheduling decisions, allowing SDFG analysis to obtain properties like throughput or buffer sizes for the scheduled graphs. Analysis times depend strongly on the size of the SDFG. SDFGs can be statically scheduled using static-order schedules. The only generally applicable technique to model a static-order schedule in an SDFG is to convert it to a homogeneous SDFG (HSDFG). This conversion may lead to an exponential increase in the size of the graph and to sub-optimal analysis results (e.g., for buffer sizes in multi-processors). We present a technique to model periodic static-order schedules directly in an SDFG. Experiments show that our technique produces more compact graphs compared to the technique that relies on a conversion to an HSDFG. This results in reduced analysis times for performance properties and tighter resource requirements.**

I. Introduction

Synchronous dataflow graphs (SDFGs) are widely used to model digital signal processing and multimedia applications [1]–[4]. Model-based design-flows (e.g., [1], [5]–[8]) model binding and scheduling decisions into the SDFG. This enables analysis of performance properties (e.g, throughput [9]) or resource requirements (e.g., buffer sizes [10]) under resource constraints. Fig. 1 shows an example of an SDFG $G(A, C)$ with four *actors* ($A = \{a_0, a_1, a_2, a_3\}$) and three *channels* ($C = \{c_0, c_1, c_2\}$). These actors communicate with *tokens* sent from one actor to another over the channels. Channels may contain tokens, depicted with a solid dot (and an attached number in case of multiple tokens) (e.g., see Fig. 2). An essential property of SDFGs is that every time an actor *fires* (executes) it consumes the same amount of tokens from its input edges and produces the same amount of tokens on its output edges. These amounts are called the *rates* (indicated next to the channel ends when the rates are larger than 1). The fixed port rates make it possible to statically schedule SDFGs.

Many SDFG analysis algorithms, e.g., throughput calculation or buffer sizing, are straightforward when a single processor platform is used. For instance, the buffer sizes can be determined by executing the SDFG according to a given schedule. However, in a multi-processor environment, SDFG analysis algorithms are not trivial because of the inter-processor communication amongst other reasons. For a multi-processor, it is possible to construct per processor to which actors of the SDFG are bound, a finite schedule that sequentially orders the actor firings and which is repeated indefinitely. Such

This work was supported in part by the Dutch Technology Foundation STW, project NEST 10346.

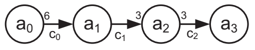


Figure 1. An example SDFG

a schedule is called a *periodic static-order schedule* (*PSOS*). PSOSs specify the order of actor firing which separates them from fully static schedules, which determine absolute start times of actors. A model-based design-flow usually uses PSOSs for an application modeled with an SDFG. In this way timing (throughput) and memory usage (buffers) can be analyzed.

There is only one technique [11] known to model PSOSs in an SDFG. This technique requires a conversion of an SDFG to a so-called *homogeneous SDFG* (HSDFG) in which all rates are equal to one [2]. Fig. 2 (without the blue edges) shows the equivalent HSDFG of our example SDFG of Fig. 1. The PSOS modeling technique of [11] sequentializes the actor firings by inserting a channel between each pair of consecutive actors in a processor schedule. At the end of each schedule, it adds a channel with one initial token from the last to the first actor in the schedule. All of these ensure an indefinite execution of the graph according to the schedules. The technique from [11] adds in total 15 channels to the HSDFG of our example graph (the blue edges in Fig. 2) to model PSOSs $s_0 = \langle a_0(a_2)^2\rangle^*$ and $s_1 = \langle (a_1)^5(a_3)^3 a_1(a_3)^3\rangle^*$.

The SDFG to HSDFG conversion can lead to an exponential increase in the size of the graph. For example, converting the SDFG of an H.263 decoder [10] to the equivalent HSDFG increases the graph size from 4 actors to 200 actors. The run-time of SDFG analysis algorithms depends amongst others on the size of the graph. As a result, the run-time of many SDFG analysis algorithms may increase drastically when modeling PSOSs in the graph using the technique from [11]. For example, the buffer sizing algorithm from [10] takes less than 1 ms on the SDFG of an H.263 decoder. Modeling a schedule into this SDFG using the technique from [11], the run-time of the same algorithm increases to 1330 ms. SDFG analysis algorithms are usually repeated more than once in an iterative design-flow. For example, the design-flow from [6] performs 8 throughput calculations to determine the right solution for an H.263 decoder. Hence, it is vital to keep the size of the schedule-extended graph as small as possible to provide a fast and practical design trajectory. There is a second drawback to the technique from [11]. The original graph structure is lost due to the conversion to an HSDFG. A single channel in an SDFG corresponds to a set of channels in the HSDFG. As a result, common buffer sizing techniques cannot find the minimal buffer size for the original SDFG. The H.263 decoder buffer sizes are for example overestimated by 43% when applying the technique of [10] to the HSDFG. Note that a conversion to an HSDFG may be required in a code generation step. However, if this conversion can be delayed until all analysis are carried out on the SDFG, then this can save a significant amount of resources (e.g., buffer space) and analysis time.

A novel technique is needed to model any PSOS in an SDFG. This technique should limit the increase in the number of actors such that analysis times do not increase too much when analyzing the SDFG with its schedules. The technique should also preserve the original graph structure as this enables accurate analysis of graph properties such as buffer sizes. This paper presents a technique that satisfies both requirements. The technique can be used in any model-based design-flow that models PSOSs into the SDFG (e.g., [1], [5]–[8]). The proposed technique can also directly be used to model PSOSs in scenario-aware dataflow graphs [12].

The remainder of the paper is structured as follows. The next section discusses related work. Sec. III sketches basic concepts. Sec. IV presents our technique to model PSOSs in an SDFG. We evaluate our technique by applying it to several realistic applications in Sec. V. Sec. VI concludes.

II. Related work

The technique from [11] is the only available technique to model PSOSs in an SDFG. As already explained, this technique may result in a long run-time for analysis algorithms and/or inaccurate results from these algorithms. Our technique alleviates both shortcomings of the technique from [11]. The work in [13] models the effect of a budget scheduler or preemptive TDMA on the temporal behavior of the SDFG, either by computing an accurate worst-case response time, or more precisely by introducing additional actors into a latency-rate model. In contrast, for non-preemptive schedules, such as PSOSs, we focus on the ordering of actor firings; their execution time remains the same. We enforce an SDFG to follow the PSOSs selected for each processor. This allows SDFG analysis to obtain properties like throughput or buffer sizes for the scheduled SDFG. Since we only use the basic components of an SDFG (e.g., actors and channels) to model schedules in an SDFG, our schedule-extended SDFG can be directly used in any model-based design-flow (e.g., [1], [5]–[8]). Ref [14] uses some new (custom) components, e.g., $if - then - else$, to model schedules in an SDFG. The common model-based design-flows do not support these new components and it is not possible to model these components by using the basic components of an SDFG. Our technique eliminates the need for any new (custom) component. As a result, any analysis technique for SDFGs is directly applicable on the schedule-extended SDFG.

III. Preliminaries

The rates in an SDFG determine how often actors have to fire with respect to each other such that the distribution of tokens over all channels is not changed. This property is captured in the *repetition vector* [1] of an SDFG ($\gamma(a)$ refers to the repetition vector value of actor a). The repetition vector

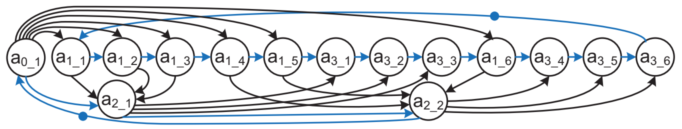


Figure 2. PSOSs s_0 and s_1 modeled in the SDFG of Fig. 1 using the technique from [11].

of the SDFG shown in Fig. 1 is equal to $(a_0, a_1, a_2, a_3) \rightarrow (1, 6, 2, 6)$. A firing of actor a leads to the consumption of tokens from its input channels and the production of tokens on its output channels. Hence, channels can contain different amounts of tokens according to the designated schedule. A state of an SDFG (represented by ω) is determined by the amount of tokens in all channels of the SDFG. We assume that the initial state of an SDFG is given by some initial token distribution ω_0. An actor can only fire if sufficient tokens are available on the channels from which it consumes. An actor that satisfies this condition in a particular state is said to be *enabled* in this state. Consistency (i.e., the existence of a repetition vector) and absence of deadlock are practically necessary conditions for SDFGs which can be verified efficiently [15], [16]. Any SDFG which is not consistent requires unbounded memory to execute or deadlocks. Therefore, we limit ourselves to consistent and deadlock free SDFGs.

When a consistent and deadlock-free SDFG is executed according to one or more PSOSs, the channels of the SDFG need bounded memories (according to Theorem 1 from [17]). The number of actor appearances in the PSOS is a fraction of its repetition vector entry. Formally, each actor a in the PSOS should appear $r \cdot \gamma(a)$ times in the PSOS ($r = \frac{u}{v}$ where $u, v \in \mathbb{N}$) and the value r is identical for all actors in the PSOS [9]. This follows from the SDFG property that firing each actor as often as indicated in the repetition vector results in a token distribution that is equal to the initial token distribution. In the paper, the term *normalized PSOS* is used to refer to a PSOS with r equal to 1. We limit ourselves in the remainder to PSOSs in which r is a *unit fraction* (i.e., $r = \frac{u}{w}$ with $u = 1$ and $w \in \mathbb{N}$), although our technique can also be directly applied to model other PSOSs (i.e., in which $u \in \mathbb{N}$).

IV. Modeling Periodic Static-Order Schedules

In this section, we introduce a technique to model PSOSs in an SDFG. Algorithm 1 encapsulates our technique, called decision state modeling (DSM). Fig. 3 depicts the corresponding SDFG of Fig. 1 which models the PSOSs s_0 and s_1 using DSM. The remainder of this section discusses different parts of the algorithm in detail. There are several reasons why an SDFG cannot model PSOSs naturally. The following subsections discuss them and illustrate how we address them.

The description of some basic functions used in Algorithm 1 is as follows. The function *AA(G, a_{new})* is responsible to include the actor a_{new} in the SDFG G. The function *AC(G, c_{new}, a_{src}, a_{dst}, $srcRate$, $dstRate$, $initTok$)* adds the channel c_{new} from the source actor a_{src} to the destination actor a_{dst}; the production (consumption) rate of a_{src} (a_{dst}) on this channel is equal to $srcRate$ ($dstRate$); this channel is initialized with $initTok$ tokens. The function *BEF(a_k,ω_j,s_i)* (*AFT(a_k,ω_j,s_i)*) returns the number of times that a_k appears before (after) state ω_j in one repetition of the PSOS s_i.

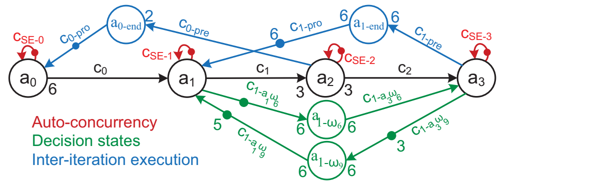


Figure 3. PSOSs s_0 and s_1 modeled in the SDFG of Fig. 1 using DSM.

A. *Auto-concurrency*

An actor $a \in A$ in an SDFG state ω can possibly be enabled multiple times simultaneously in ω. This property is called auto-concurrency. The firings related to actor a should occur sequentially according to the PSOS to which actor a belongs. This sequential execution can be enforced by adding a self-edge with one initial token to actor a (Line 1 in Algorithm 1). In Fig. 3, channels $c_{SE-0} - c_{SE-3}$ (shown in red) are used to prevent any auto-concurrency in the SDFG of Fig. 1.

B. *Inter-iteration execution*

Enabled actors in a PSOS belonging to the next PSOS iteration prevent the execution of the SDFG from following the given PSOS; lines 4-8 in Algorithm 1 are used to control this undesirable actor enabling. This part of the algorithm adds (per PSOS) one actor and two channels to create a dependency between the last and first actor appearing in the PSOS. The added components limit, within one PSOS iteration, the firing of the first actor in the PSOS (i.e., a_F) to the count of actor a_F (i.e., $CNT(a_F, s_i)$) in one iteration of the PSOS s_i. The function $CNT(a_F, s_i)$ in DSM returns the count of the actor a_F in one iteration of the PSOS s_i. The next iteration of the PSOS s_i can only commence if the last actor in PSOS s_i (i.e., a_L) fires $CNT(a_L, s_i)$ times in one iteration of the PSOS s_i. In other words, the next iteration of a PSOS can only commence after the completion of the current iteration of this PSOS. In Fig. 3, actor a_{0-end} and channels c_{0-pre} and c_{0-pro} are added to prevent any inter-iteration execution in PSOS s_0. Actor a_{1-end} and channels c_{1-pre} and c_{1-pro} are added to prevent any inter-iteration execution in schedule s_1.

Algorithm 1: Decision State Modeling (DSM)

```
input : SDFG $G(A, C)$, PSOSs $\{s_0, \cdots, s_n\}$
output: $G$ extended with schedules $\{s_0, \cdots, s_n\}$
1  add a self edge with 1 initial token for each $a \in A$
2  $\{s'_0, \mu_0, \cdots s'_n, \mu_n\} \leftarrow$ normalize$(G, \{s_0, \cdots, s_n\})$
3  for $i \leftarrow 0$ to $n$ do
4      $a_L$ := last actor in $s_i$
5      $a_F$ := first actor in $s_i$
6      AA$(G, a_{i-end})$
7      AC$(G, c_{i-pre}, a_L, a_{i-end}, 1, $CNT$(a_L, s_i), 0)$
8      AC$(G, c_{i-pro}, a_{i-end}, a_F, $CNT$(a_F, s_i), 1, $CNT$(a_F, s_i))$
9      $\Omega \leftarrow$ getDecisionStates$(G, s'_i, \{s'_0, \cdots, s'_n\} \setminus s'_i)$
10     $\Omega \leftarrow$ reduceDecisionStates$(\Omega)$
11     $\Omega \leftarrow$ foldDecisionStates$(\Omega, \mu_i)$
12     foreach $\omega_j \in \Omega$ do
13         AA(G, $a_{i-\omega_j}$)
14         foreach $a_k \in \Delta_j$ do
15             if $a_k$ is the actor of choice then
16                 AC$(G, c_{i-a_k\omega_j}, a_k, a_{i-\omega_j}, 1, $CNT$(a_k, s_i), $AFT$(a_k, \omega_j, s_i))$
17             else
18                 AC$(G, c_{i-a_k\omega_j}, a_{i-\omega_j}, a_k, $CNT$(a_k, s_i), 1, $BEF$(a_k, \omega_j, s_i))$
```

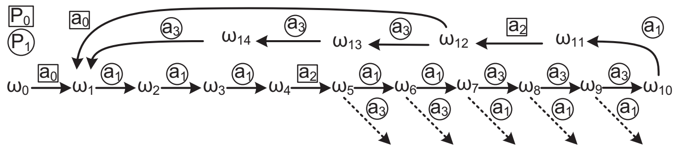


Figure 4. The state space of the SDFG of Fig. 1 when PSOSs $s_0 = \langle a_0(a_2)^2\rangle^*$ and $s_1 = \langle (a_1)^5(a_3)^3a_1(a_3)^3\rangle^*$ are used.

C. *Decision states*

1) Concept: The state space when executing our example SDFG using the PSOSs s_0 and s_1 is visualized in Fig. 4. In this figure, the actors mapped on processor P_0 (P_1) are surrounded by a square (circle). Auto-concurrency and inter-iteration execution are excluded using the constructs introduced in Sec. IV-A and Sec. IV-B respectively. The periodic behavior of the PSOSs is obvious from the state space. There are some states in which more than one actor is enabled ($\omega_5 - \omega_9$) on one processor. In such a situation, the execution related to those actors can deviate from the specified PSOS. We call a state with more than one actor enabled in one processor a *decision state* of this processor. The finite set Ω contains these decision states of this processor. One of the enabled actors in a decision state ω_j, in line with the given PSOS s_i, should be selected to get fired. We call all enabled actors of s_i in ω_j *opponent actors*; we call the actor that should be executed in ω_j the *actor of choice* of ω_j. The finite set Δ_j represents the opponent actors in the decision state $\omega_j \in \Omega$. One member of the set Δ_j is the actor of choice in ω_j; in the paper, we denote that actor of choice with $a_c \in \Delta_j$.

Lines 9-18 in DSM show how we deal with uncertainty due to decision states. In the algorithm $n+1$ ($n \in \mathbb{N}_0$) is the number of the processors (or input PSOSs). DSM models the given PSOSs one-by-one iteratively. The ordering of PSOSs in DSM does not have any impact on the final outcome. In each iteration of the for-loop in line 3, we enforce the execution of the actors in the current schedule of interest (i.e., schedule s_i) to follow schedule s_i. The next sub-section explains how decision states of the schedule of interest are extracted. DSM for each $\omega_j \in \Omega$ extracted from s_i adds an actor ($a_{i-\omega_j}$) and one channel between the new actor $a_{i-\omega_j}$ and each opponent actor in the set Δ_j (lines 14-18 in Algorithm 1). In our example, these elements are shown in green in Fig. 3. In practice, the elements added in each decision state (e.g., ω_j) postpone the execution of the actors in $\Delta_j \setminus \{a_c\}$ to the state after decision state ω_j. Hence, a_c (i.e., the actor of choice) is the only actor which can be fired in the state ω_j.

2) Decision state identification: Algorithm 2 shows our proposed technique to detect all decision states. In this algorithm, s_c is the PSOS for which we want to determine the decision states. Assume s_c is a PSOS for the actors mapped on processor P_c. Schedules $s_{o1} \cdots s_{on}$ are PSOSs for the other actors of the SDFG mapped on the other processors (with $P_{o1} \cdots P_{on}$ as the other processors). In Algorithm 2, the input schedules are normalized PSOSs. The function *normalize* (in line 2 of Algorithm 1) normalizes the input PSOSs. The function returns the normalized PSOSs along with their normalization factors. The normalized PSOS s'_x can be achieved by repeating μ_x times the input PSOS s_x (i.e., $s'_x = (s_x)^{\mu_x}$). μ_x is the normalization factor of s_x and can be calculated by dividing the repetition vector entry of an arbitrary actor in s_x by the count of that actor in the PSOS s_x (in our example, μ_0 and μ_1 are equal to 1).

An actor in the schedule of interest s_c could be affected by the execution of an actor in the other schedules as well as another actor in s_c. Processors can run at different clock rates; these differences and inter-processor dependencies cause variation in the amount of tokens on the *inter-processor channels* originating from the actors mapped on the other processors to the actors mapped on the processor of interest (i.e., P_c). The amount of tokens on the input channels of an actor determines whether an actor is enabled or not. Our technique can determine any possible actor enabling when executing s_c by considering the maximum amount of tokens on all inter-processor channels. Each iteration of s_c requires that the actors mapped on the other processors are fired up-to at most their repetition vector entry values. Hence, only executing one iteration of the other schedules $s_{o1} \cdots s_{on}$ is enough to provide sufficient tokens on inter-processor channels entering to the actors mapped on processor P_c. Subsequent iterations of the other schedules $s_{o1} \cdots s_{on}$ are possible; this may enable an actor in s_c to be enabled more than its designated amount in one iteration of s_c. The inter-iteration prevention constructs introduced in Sec. IV-B are used to control this undesired actor enabling. So, we only extract decision states within one iteration of the normalized schedule. Also, DSM does not impose any limitation between PSOSs; PSOSs can independently be iterated if the dependencies in the SDFG allow that. We allow the actors on the other processors to be executed (according to their schedules) as much as they can. The execution of the actors on the other processors will stop at one point either due to their dependency on the actors on the processor P_c or because one iteration of their schedule is completed. The state of the SDFG needs to be preserved to follow the subsequent execution of the actors. This maximal execution of the actors on the other processors is represented by the function *maxExec* in Algorithm 2. After this maximal execution, the amount of tokens on the inter-processor channels entering into the actors on the processor P_c determines any possible enabled actor. The current state (represented by ω_j) will be added to the decision state set (Ω) if more than one actor on the processor P_c is enabled at this state (line 4 in Algorithm 2). All enabled actors will be recorded as opponent actors of the state ω_j (line 5 in Algorithm 2). The execution of the actors on the processor P_c is continued by executing the enabled actor in line with s_c in order to determine all possible decision states (line 6 in Algorithm 2). The function $fire(G, s_c[i])$ executes once the i^{th} actor in s_c. The maximal execution followed by decision state identification will be iterated to execute one iteration of s_c. In the end, the set Ω contains all possible decision states when executing s_c. In the SDFG of Fig. 1, five consecutive decision states ($\Omega = \{\omega_5 \cdots \omega_9\}$) exist for s_1 and no decision state exists for s_0 (see Fig. 4).

3) Redundant decision states: It is possible to have several consecutive decision states which are postponing the firing of an actor to several states later. For example, three consecutive decision states ($\omega_7 - \omega_9$) exist in Fig. 4; the added components in ω_7 postpone the sixth firing of a_1 to ω_8; the added components in ω_8 postpone the sixth firing of a_1 to ω_9; and so on. The latest decision state in the sequence of decision states $\omega_7 - \omega_9$ is enough to postpone the firing of a_1 to ω_{10}. Hence, the decision states $\omega_7 - \omega_8$ are redundant and can be removed from the decision state set Ω. The function *reduceDecisionStates* is responsible for removing redundant decision states. Note that it would be possible to perform this reduction during the decision state identification step. This reduction can remove a significant amount of extra components in the final SDFG. Decision state ω_5 is also redundant according to our optimization. So, only two decision states ω_6 and ω_9 are necessary to model s_1 in the SDFG of Fig. 1.

Algorithm 2: Get Decision States

```
input  : SDFG $G$, PSOS $s_c$, PSOSs $\{s_{o1}, \cdots, s_{on}\}$
output: Decision state set $\Omega$
1 for  $i \leftarrow 1$ to sizeof($s_c$) do
2     maxExec(G, $\{s_{o1}, \cdots, s_{on}\}$)
3     if sizeof(enabledActors(G, $s_c$)) >1 then
4         $\Omega \leftarrow \Omega \cup \omega_j$    /* $\omega_j$ is the current state */
5         $\Delta_j \leftarrow$ enabledActors(G, $s_c$)
6     fire(G, $s_c[i]$)
```

4) Decision state folding: In Algorithm 1, the input PSOSs are normalized to find all decision states. The normalization of PSOSs is required to explore all (sufficient) states of an SDFG. Consider PSOSs $s_2 = \langle a_0 \rangle^*$ and $s_3 = \langle a_2\ a_1 \rangle^*$ for our second example SDFG in Fig. 5. To obtain normalized PSOSs, μ_2 and μ_3 must be equal to 3 and 4 respectively. This leads to the following normalized PSOSs: $s'_2 = \langle (a_0)^3 \rangle^*$ and $s'_3 = \langle (a_2\ a_1)^4 \rangle^*$. $\underbrace{\binom{a_2}{-}\binom{a_1}{a_2}}_{1^{th}\quad 2^{th}}\underbrace{\binom{a_2}{a_1}\binom{a_1}{a_2}}_{3^{th}\quad 4^{th}}\underbrace{\binom{a_2}{a_1}\binom{a_1}{a_2}}_{5^{th}\quad 6^{th}}\underbrace{\binom{a_2}{-}\binom{a_1}{-}}_{7^{th}\quad 8^{th}}$ shows the corresponding execution of s'_3. In construct $\binom{a_x}{a_y}$, a_x is the enabled actor in line with the PSOS and a_y is the other enabled actor if any at all. In this execution, the 1^{st}, 3^{th}, 5^{th} and 7^{th} states are similar in behavior. In other words, the actor a_2 should be fired in all of those states.

Modeling a repetitive behavior for a PSOS s_i, also models its normalized PSOS (i.e., $s'_i = (s_i)^{\mu_i}$). By considering this fact, we can merge decision states appearing in all μ_i repetitions of s_i. We call this optimization *decision state folding* (line 11 in Algorithm 1). Folding groups the similar states. In our example, the 1^{st}, 3^{th}, 5^{th} and 7^{th} states are grouped and represented with one state. Similar state grouping can be performed for the 2^{th}, 4^{th}, 6^{th} and 8^{th} states. So, the above execution shrinks to $\binom{a_2}{a_1}\binom{a_1}{a_2}$. If there is a decision state in any of the similar states in the original execution, a decision state will be placed in the substitution state of those states. In practice, a decision state in a state of the new folded execution will be considered as a decision state for each of the equivalent states in the original execution. This cannot violate the execution according to the input PSOS because DSM only ensures the execution of the actor of choice in a decision state. This optimization could reduce the number of decision states up to μ_i times in a normalized PSOS s'_i. The decision state in the last state is ignored thanks to our inter-iteration execution prevention (which is explained in Sec. IV-B). In our second

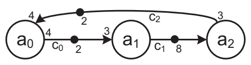


Figure 5. An example SDFG

example, decision state folding reduces the number of decision states from 5 to 1 for s_3.

5) Enforcing a schedule in decision states: In our first example SDFG, only two actors are enabled in decision state ω_6 (i.e., $\Delta_6 = \{a_1, a_3\}$) (see Fig. 4). Actor a_1 is the actor of choice in decision state ω_6 and actor a_3 is the only opponent actor whose execution should be postponed to the state after state ω_6. DSM adds actor $a_{1-\omega_6}$ and channels $c_{1-a_1\omega_6}$ and $c_{1-a_3\omega_6}$ to the graph in decision state ω_6. DSM also adds actor $a_{1-\omega_9}$ and channels $c_{1-a_1\omega_9}$ and $c_{1-a_3\omega_9}$ to the graph for the other decision state ω_9.

The actor $a_{1-\omega_9}$ is added to enforce the firing of a_3 in decision state ω_9 and postpone the execution of a_1 to the subsequent state. The actor $a_{1-\omega_9}$ is only responsible for decision state ω_9 and it fires only once in an iteration. This means that its value in the repetition vector of the new graph (i.e., Fig. 3) is one. The production and consumption rates of the ports of the actor $a_{1-\omega_9}$ should be set to a value to preserve the consistency of the SDFG; for this purpose, the port rates of actor $a_{1-\omega_9}$ on its channels (i.e., $c_{1-a_1\omega_9}$ and $c_{1-a_3\omega_9}$) are set to 6. The added dependency channels from the newly added actor in decision state ω_j (e.g., $a_{1-\omega_9}$ in decision state ω_9) to the opponent actors which are not the actor of choice (e.g., a_1 in decision state ω_9) only provide enough tokens for their execution in states $\omega_0 - \omega_{j-1}$ (e.g., 5 tokens for a_1 in states $\omega_0 - \omega_8$); these actors cannot be enabled due to the lack of tokens in the newly added channels in the corresponding decision state (e.g., there will be no token in channel $c_{1-a_1\omega_9}$ in decision state ω_9). Hence only the actor of choice amongst the opponent actors of a decision state will be enabled in that state (e.g., only a_3 can fire in decision state ω_9). The firing of the postponed actors in a decision state (e.g., decision state ω_j) will not depend on the newly added actor in the decision state (i.e., $a_{i-\omega_j}$) after firing of the actor of choice in ω_j. For example, there will be 6 tokens in channel $c_{1-a_3\omega_9}$ after firing of actor a_3 (i.e., the actor of choice) in decision state ω_9; hence, the actor $a_{1-\omega_9}$ can immediately fire and its execution will provide sufficient tokens for later firings of actor a_1. So, the postponed actor in decision state ω_9 will not be dependent on actor $a_{1-\omega_9}$ for its later execution in the current iteration of the PSOS s_1.

The firing of actor a_3 after decision state ω_9 produces 3 tokens in channel $c_{1-a_3\omega_9}$ and the firing of actor a_1 after decision state ω_9 consumes 1 token from channel $c_{1-a_3\omega_9}$; as a result, the amount of tokens in the new channels are reset to the initial values at the end of one iteration of the schedule s_1. Hence, the periodic behavior is also achievable for the added components. The components added in decision state ω_6 show similar behavior as the components added in decision state ω_9.

D. Correctness of DSM

The following theorems state the correctness of DSM in modeling a single PSOS for a subset of the actors of the SDFG. If we can model a single PSOS in the SDFG, then we can also model multiple PSOSs by simply applying the result multiple times. The proofs of the theorems are available through [18].

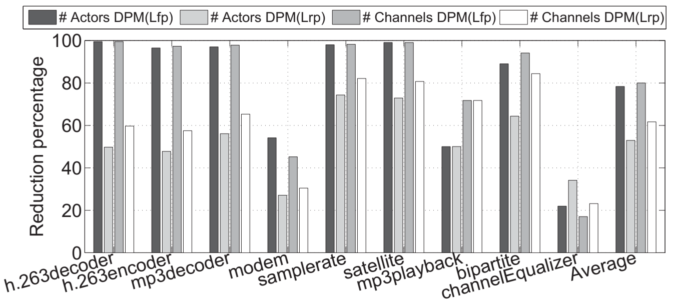


Figure 6. Reduction in the size of the schedule-extended graphs when using DSM in contrast to the HSDFG-based technique (Higher is better). Schedules are generated by list forward priorities (Lfp) and list reverse priorities (Lrp).

Theorem 1. *Any execution possible in the schedule-extended SDFG should respect the PSOS.*

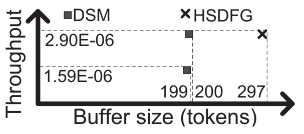
Theorem 2. *Any execution possible in the original SDFG that already satisfies the PSOS, should also be a valid execution for the schedule-extended SDFG.*

V. Experimental results

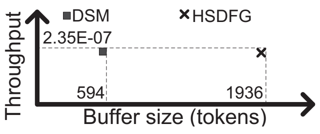
We used a set of DSP and multimedia applications to assess our DSM technique. The following SDFGs are extracted from realistic applications: modem [1], sample-rate converter [1], satellite receiver [19], mp3playback [20], channel equalizer [21], H.263 decoder [10], H.263 encoder [22], and MP3 decoder [10]. We also consider the bipartite SDFG [19] which is a commonly used artificial SDFG.

A PSOS determines the actor firing order and as such it influences the enabled actors in a state; as a result, the number of decision states can be different for different PSOSs. The size of the schedule-extended graph using DSM depends on the number of decision states in the given schedules. So, the compactness of the schedule-extended graph depends on the input schedule which should be modeled. We use the common list scheduler [23] to determine the PSOSs for the applications. We use two different variations of list scheduling to verify DSM in different situations. The first list schedule uses forward priorities (Lfp) and the second one uses reverse priorities (Lrp). Actors closer to the inputs of the graph have higher priority in the Lfp schedules compared to actors closer to the outputs of the graph and vice-versa in Lrp schedules.

Fig. 6 shows the reduction percentage in the size of the schedule-extended graph when using DSM in contrast to the HSDFG-based technique. Using schedules generated by Lfp, the number of decision states is less than when Lrp is used, except in the channel equalizer and mp3playback applications. By using Lfp scheduling, actors closer to inputs have higher priority compared to actors closer to outputs. This leads to consecutive execution of an actor followed by consecutive execution of another actor with lower priority and so on. Thanks to our optimization in DSM, considering only one decision state before a context switch will be sufficient (e.g., decision state ω_9 in Fig. 4) and the number of decision states can be reduced significantly. Usually actors closer to outputs are dependent on actor closer to inputs in an SDFG; this dependency can prevent an actor from being executed consecutively in a graph scheduled by Lrp. As a result of that, the



(a) H.263 decoder



(b) MP3 decoder

Figure 7. Pareto space of schedule-extended graphs modeled by DSM and HSDFG-based techniques (the scales of two graphs are different).

number of context switches in a graph scheduled by Lrp will typically be larger compared to Lfp. Hence, the effectiveness of the decision state optimization in DSM reduces and extra elements are required to model the schedules in the graph. The exceptions in the channel equalizer and mp3playback are due to the existence of a cycle in the SDFG; the cycle can increase the number of context switches in the schedule and as a result, Lfp could result in the same or a higher amount of decision states in DSM compared to Lrp. Important, however, is that in our experiments, DSM always outperforms the HSDFG-based technique regardless of the input schedule. The number of actors (channels) using DSM is 66% (71%) lower compared to the HSDFG-based technique on average, 99% (99%) lower in the best-case and 28% (20%) lower in the worst-case observed in our experiments. Besides the compactness of the schedule-extended graph, DSM preserves the original structure of an SDFG which is not guaranteed for the state of the art technique.

To further analyze the effectiveness of DSM, we applied a buffer sizing algorithm from [10] on the schedule-extended SDFGs of the H.263 decoder and MP3 decoder applications. The H.263 decoder is mapped on a platform with two processors. The actor *vld* and *iq* are mapped on the first processor with a PSOS $\langle vld(iq)^{99}\rangle^*$ and the actor *idct* and *mc* are mapped on the second processor with a PSOS $\langle (idct)^{99}mc\rangle^*$. The analysis time for buffer sizing on the schedule-extended H.263 decoder is less than 1 ms when using DSM to model the schedules. The same analysis lasts for 1330 ms when using the technique from [11] to model the same schedules in the same graph. Fig. 7(a) shows the complete design space (Pareto space) of throughput and buffer size when modeling the schedule with DSM and the HSDFG-based technique [11]. A single channel in an SDFG corresponds to a set of channels in the equivalent HSDFG. As a result, the buffer sizing technique cannot find the minimal buffer size when applying it on the equivalent HSDFG. Our experiments show these inaccuracies. Applying buffer sizing on the graph which models the schedules using the technique from [11] results in 43% overestimation in required buffer space compared to applying the same buffer sizing technique on the graph which models the same schedules when using our technique. Fig. 7(b) shows results for the MP3 decoder. We used the mapping and scheduling from [12] which maps the MP3 decoder on a platform with 3 processors. The analysis time on the graph which models the schedule using our technique is 594 ms while 141610 ms is required to perform the same analysis on the graph using the technique from [11]. Using the technique from [11] results in 2.26 times overestimation in buffer size compared to using our technique.

Modeling a PSOS in an SDFG using DSM requires executing one complete SDFG iteration. The number of states in one iteration could be exponential in the number of actors in the graph. However, for all real-world SDFGs used in our experiments, the execution time of the DSM is below 1 ms.

VI. Conclusion

We presented a technique, DSM, to model periodic static-order schedules directly in an SDFG. The resulting graphs are much smaller (often much less than half the size) than graphs resulting from the state of the art technique that first converts an SDFG to an HSDFG. This results in a speed-up of performance analysis. Computing the trade-off between buffering and throughput for multi-processor implementations, for example, becomes several orders of magnitude faster. Moreover properties like buffer sizes can be analyzed more accurately. For future work, we would like to investigate to further optimize the models for some specific scheduling classes, e.g., single appearance schedules.

References

[1] S. S. Bhattacharyya *et al.*, "Synthesis of embedded software from synchronous dataflow specifications," *Journal of VLSI Signal Processing*, vol. 21, pp. 151–166, 1999.

[2] S. Sriram *et al.*, *Embedded Multiprocessors: Scheduling and Synchronization*, 2nd ed. CRC Press, 2009.

[3] P. Poplavko *et al.*, "Task-level timing models for guaranteed performance in multiprocessor networks-on-chip," CASES. ACM, 2003, pp. 63–72.

[4] M.-Y. Ko *et al.*, "Compact procedural implementation in DSP software synthesis through recursive graph decomposition," SCOPES. ACM, 2004, pp. 47–61.

[5] A. Bonfietti *et al.*, "Throughput constraint for synchronous data flow graphs," CPAIOR. Springer-Verlag, 2009, pp. 26–40.

[6] S. Stuijk *et al.*, "Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs," DAC. ACM, 2007, pp. 777–782.

[7] W. Liu *et al.*, "Efficient SAT-based mapping and scheduling of homogeneous synchronous dataflow graphs for throughput optimization," RTSS. IEEE, 2008, pp. 492–504.

[8] Y. Yang *et al.*, "Automated bottleneck-driven design-space exploration of media processing systems," DATE. ACM, 2010, pp. 1041–1046.

[9] A. Ghamarian *et al.*, "Throughput analysis of synchronous data flow graphs," ACSD. IEEE, 2006, pp. 25–36.

[10] S. Stuijk *et al.*, "Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs," *IEEE Trans. on Computers*, vol. 57, no. 10, pp. 1331–1345, 2008.

[11] N. Bambha *et al.*, "Intermediate representations for design automation of multiprocessor DSP systems," *Design Automation for Embedded Systems*, vol. 7, no. 4, pp. 307–323, 2002.

[12] M. Geilen *et al.*, "Worst-case performance analysis of synchronous dataflow scenarios," CODES+ISSS. ACM, 2010, pp. 125–134.

[13] M. H. Wiggers *et al.*, "Monotonicity and run-time scheduling," EMSOFT. ACM, 2009, pp. 177–186.

[14] H. huang Wu *et al.*, "A model-based schedule representation for heterogeneous mapping of dataflow graphs," HCW. IEEE, 2011, pp. 66–77.

[15] S. Bhattacharyya *et al.*, *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.

[16] E. Lee *et al.*, "Synchronous data flow," *Proceeding of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[17] M. Geilen *et al.*, "Minimising buffer requirements of synchronous dataflow graphs with model checking," DAC. ACM, 2005, pp. 819–824.

[18] M. Damavandpeyma *et al.*, "Modeling static-order schedules in synchronous dataflow graphs," TU Eindhoven, Tech. Rep. ESR-2012-01, March 2012.

[19] S. Ritz *et al.*, "Scheduling for optimum data memory compaction in block diagram oriented software synthesis," ICASSP. IEEE, 1995, pp. 2651–2654.

[20] M. H. Wiggers *et al.*, "Efficient computation of buffer capacities for cyclo-static dataflow graphs," DAC. ACM, 2007, pp. 658–663.

[21] A. Moonen *et al.*, "Practical and accurate throughput analysis with the cyclo static dataflow model," MASCOTS. IEEE, 2007, pp. 238–245.

[22] H. Oh *et al.*, "Fractional rate dataflow model for efficient code synthesis," *Journal of VLSI Signal Processing*, vol. 37, pp. 41–51, 2004.

[23] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*, 1st ed. McGraw-Hill, 1994.