Modeling Pipelined Application with Synchronous Data Flow Graphs

Marco Lattuada, Fabrizio Ferrandi
Politecnico di Milano – Dipartimento di Elettronica, Informazione e Bioingegneria
Via Ponzio 34/5, Milan, Italy
{*lattuada,ferrandi*}*@elet.polimi.it*

***Abstract*—Streaming applications can efficiently exploit multiprocessors architectures by means of pipelined parallelism, but designing this type of applications can be an hard task. Different subproblems have indeed to be solved: partitioning, mapping, scheduling and pipeline stage assignment. For this reason, high level abstraction models are adopted during design flow since they simplify this process by hiding most of the architectural details. Synchronous Data Flow (SDF) graphs, widely adopted to describe streaming applications, naturally model only their partitioning, so they usually have to be integrated with other types of representations.**

In this paper *Pipelined Application Modeling (PAM)*, a methodology to create a Synchronous Data Flow graph describing all the aspects of a pipelined application, is presented. The methodology starts from the SDF graph describing the partitioning of the application and enriches it with new actors and channels detailing the mapping, the scheduling and the pipeline stage assignment of the considered solution. The obtained SDF graph, describing all the aspects of the solution in a formal and compact way, facilitates the evaluation of different solutions during design space exploration.

I. Introduction

Multiprocessor Systems on Chip have become the de-facto standard in embedded systems, but, in order to fully exploit the computational power they provide, design of parallel applications is required. Streaming applications (i.e., applications where a set of elaborations is applied in sequence on an input data stream) usually can be easily parallelized by means of pipelining. In this programming paradigm, the whole functionality is divided into tasks, each of which elaborates in parallel the result produced by the previous task in the previous elaboration step. All the tasks start their execution at the same time and wait for the end of all the other tasks before starting a new iteration.

Despite the simplicity of the pipelining paradigm, developing this type of applications can be a very complex activity, since several decisions have to be taken: how the application is decomposed in tasks (*partitioning*), to which processing element each task is assigned (*mapping*), in which order tasks assigned to the same processing element are executed (*scheduling*), and, in case of a pipelined application, to which pipeline stage each task is assigned (*stage assignment*). To speed-up the design process high abstraction level representations have been introduced. Streaming applications are in particular well described by Synchronous Data Flow (SDF) graphs [1]: vertices (*actors*) represent the different functionalities in which the application is decomposed while edges (*channels*) represent the data communications between them. One of the main issues of adopting high abstraction level descriptions is the distance between the abstract model and the final implementation since the former cannot model all the details of a system. Indeed, not only the filling of this gap can require further and more complex analyses, but if the high abstraction level representations hide too many aspects of the final implementations, their analysis can produce misleading results reducing the quality of the final solution. The SDF graphs well model a partitioning solution and the data exchanged among different tasks, but, even if they are a good starting point for further analyses, they cannot describe in a such immediate way the mapping, the scheduling and the stage assignment. For this reason, several solutions to formally describe this information by enriching SDF graphs have been proposed. These solutions however include only part of the missing information (e.g., [2]) or describe it by means of new representations which have to be coupled with application SDF graphs (e.g., [3], [4]). While these are a well known representation used in several methodologies and tools, none of the representations proposed to integrate them has had the same success and has been commonly accepted, potentially preventing integration of different analyses and design algorithms in the same design flow.

In this paper *Pipelined Application Modeling (PAM)*, a methodology to fully integrate a design solution directly in the SDF graph of an application, is proposed. With respect to the *Decision State Modeling* presented in [2], the methodology does not model only the partial scheduling of the application, but also the mapping and the pipeline stage assignment. The modeling is obtained by enriching the initial SDF graph with new actors and channels representing the particular design solution. Since the final representation of the design solution is still a SDF graph, all the techniques developed for the analysis and the manipulation of these graphs can be still adopted. For example, it will be shown how existing SDF analysis frameworks like SDF^3 [5] can be coupled with the proposed methodology to fully evaluate all the different solutions during design space exploration of a pipelined application.

This paper is organized as follows. Section II presents the related works. Section III introduces some preliminary definitions used in Section IV where the proposed methodology is described. Section V presents a case study showing how the proposed methodology can be exploited while Section VI draws the conclusions of this paper.

Research partially funded by the ESA's Networking/Partnering Initiative (NPI) under contract No. 4000100797.

II. Related Work

Several types of representations have been proposed to formally describe a complete specification of a pipelined application and so to allow its analysis, optimization and synthesis. Navarro et al. [6] presented an analytical model to describe such type of applications based on queueing theory. The results of the analysis of these models are then used to optimize the application through collapsing of pipeline stages and dynamic scheduling.

SDF graphs [1] can be considered as a good candidate to represent a pipelined application to be analyzed, in particular because they can be statically scheduled, reducing the application runtime overhead [7]. For example, Chan et al. [4] presented a methodology to minimize the size of the buffers in pipelined applications. They formulated a two-level heuristic which minimizes buffers by addressing at the same time mapping and stage assignment problems. The produced solution however has to be represented in an informal way, since SDF graphs are not able to natively describe it. Some techniques have been proposed to overcome these expressiveness limits by means of new types of data flow graphs or new programming languages. Damavandpeyma et al. [2] proposed to enrich a SDF graph with new actors and channels to include the scheduling solution directly in the graph. The added actors force the serialization of activation of actors which can run in parallel but that have been assigned to the same processing element. The enriched graph describes only periodic static-order schedule, i.e., the scheduling is specified only for each single processor and no information about inter processors synchronization is added. Both scheduling and mapping are instead described in the *Interprocessor Communication SDF Graphs* (IPC graphs) proposed by Bambha et al. [8]. The graphs are built starting from a design solution and describe the sequence of actor activations for each processing element and the communications among them. The graph is not an enriched version of the application SDF graph, but it is built from scratch: in this way some information (e.g., characteristics of the channels connecting actors mapped on the same processing element) of the analyzed application is lost. Moreover, initial SDF graphs have to be transformed into Homogeneous ones, so size of the graphs to be analyzed can potentially explode.

The mapping and the processors synchronization are also described in the *Dataflow Schedule Graphs* proposed by Wu et al. [3]. This type of model however, even if based on Data Flow graphs has still to be coupled with SDF graphs in order to provide a full representation of a design solution.

Examples of languages adopted to describe pipelined applications are StreamIt [9] and OpenMP, extended as proposed by Pop et al. [10]. The former is a high level java-based language proposed for the design of streaming applications that allows only to describe the structure of a SDF graph without mapping nor scheduling information. In a similar way, the latter can be used to explicitly represent consumer-producer relationships and tasks synchronization. The topology of a SDF graph can be computed starting from these relationships, but this extension does not allow to specify mapping and scheduling solutions.

The methodology proposed in this paper overcomes the limits of the previous ones enriching the SDF graph not only with scheduling information but also with mapping and stage assignment solutions preserving at the same time all the application information. Since the enriched graphs are still compliant with the SDF model, all the methodologies and frameworks available for SDF graphs can be used to analyze, refine and implement the represented design solution.

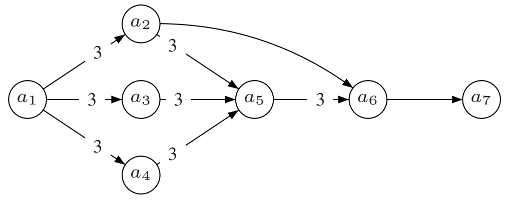

Fig. 1. Example of a SDF graph.

III. Preliminaries

In this section some notations about SDF graphs which will be used in Section IV are introduced, then the *Decision State Modeling* technique [2], which is extended in the first part of the proposed methodology, is briefly described.

A SDF graph is defined as a directed graph $G = (A, C)$ where each node $a_j \in A$ represents one of the processes composing the application while each edge $(a_j, a_k) \in C$ represents the communication channel between a producer process a_j and a consumer process a_k. A scheduling solution for a SDF graph is composed of a *periodic static-order schedule* (PSOS) s^i for each processing element i. Each s^i is a sequence specifying the order in which the actors mapped on i must be activated. The following functions will be used to analyze and manipulate the SDF graphs:

- $\texttt{R}(a_j, a_k)$: returns the number of tokens consumed by a_k from channel (a_j, a_k) at each its activation;
- $\texttt{BEF}(a_j, t, s^i)$: returns the number of activations of a_j in PSOS s^i before its t-th element;
- $\texttt{AFT}(a_j, t, s^i)$: returns the number of activations of a_j in PSOS s^i after its t-th element;
- $\texttt{CNT}(a_j, s^i)$: returns the number of activations of a_j in PSOS s^i; note that $\texttt{BEF}(a_j, t, s^i) + \texttt{AFT}(a_j, t-1, s^i) = \texttt{CNT}(a_j, s^i)$;
- $s^i[t]$: returns the t-th actor scheduled in PSOS s^i;
- $\texttt{P}(a_j)$: returns the pipeline stage to which a_j has been assigned;
- $\texttt{AA}(a_j)$: adds actor a_j to the SDF graph;
- $\texttt{AC}(a_j, a_k, w, r, d)$: adds channel (a_j, a_k) to the SDF graph and sets the number of produced, consumed and initial tokens to w, r and d respectively;
- $\texttt{AT}(a_j, a_k, n)$: adds n initial tokens to channel (a_j, a_k).

The SDF graph which will be used as example to show the application of the proposed methodology is presented in Figure 1. The number of tokens produced and consumed on each edge is one, when not differently specified by labels on the edge itself.

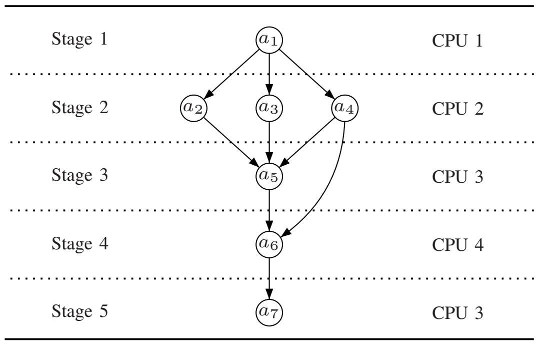

Fig. 2. Example of a design solution to be represented.

An example of design solution for the application described by SDF graph of Figure 1 is summarized in Figure 2: a_2, a_3 and a_4 have been assigned to the same pipeline stage, each of the other actors to a different pipeline stage. Moreover, stage 3 and stage 5 have been assigned to the same processing element and finally $s^2 = (a_4, a_3, a_2)$ and $s^3 = (a_5, a_7)$. Note that s^3 sets the execution order of actors assigned to different pipeline stages that have been mapped to the same processing element and that for this reason have to be sequentially executed.

Decision State Modeling (DSM) [2] has been proposed to include PSOSs directly in SDF graphs. Note that all the activations of an actor must be part of the same PSOS. A self-edge is added to all the actors to prevent auto-concurrent activations (i.e., overlapping activations of a same actor) and channels are added to prevent overlapping of different executions of the same PSOS. The evolution of the state of the SDF graph is then analyzed to identify the states, reachable from the initial one running the given PSOSs, where more than one actor assigned to the same processing element can be activated (*decision states*). Given a decision state ω^i_t, where index i identifies the considered processing element and t counts the number of states from the initial, the set Δ^i_t of actors that can be activated in that state are called *opponent actors*. The PSOS of the processing element specifies which actor a_j (*actor of choice*) has to be executed in that state. DSM forces the execution of a_j by adding a new actor $a_{\omega^i_t}$, a new channel $(a_j, a_{\omega^i_t})$ and a new channel $(a_{\omega^i_t}, a_k)$ for each $a_k \in \Delta^i_t - \{a_j\}$ and by adding the opportune initial tokens on the new channels. In this way, DSM forces the serialization between a_j and the other actors that, even if ready, have to be executed only after the end of a_j activation. Finally DSM applies some optimizations to the produced SDF graph to reduce the number of added actors such as collapsing the actors that postpone the activation of a same actor.

IV. Pipelined Application Modeling

The proposed methodology aims at formally describing a pipelined application by enriching the corresponding SDF graph. There are not any particular limitations on the analyzed application nor on the architecture running it. The methodology is not limited to the analysis of Homogeneous SDF graphs, so transformation in Homogeneous SDF graph is not required. On the contrary some other preprocessing of the analyzed SDF graph can be necessary if the analyzed solution has some particular characteristics. In the solution to be represented, all the activations of the same actor have to be mapped on the same processing element: if they are mapped on different processing elements, a replica of the actor for each of them has to be created. In a similar way, multicore processing elements and so multi-threaded solutions can be described by considering each core as a different processing element. Finally, differently from [4], the convexity of the mapping solution is not required (i.e., two actors can be mapped on the same processing element even if one of the paths which connect them crosses an actor mapped on a different processing element).

Algorithm 1: Pipelined Application Modeling (PAM).

```
Input  : G(A,C), PSOSs $\{s^1, s^2, \ldots, s^n\}$, P
Output: Modified G(A,C)
foreach auto-concurrent $a_i$ do
    AC($a_i, a_i, 1, 1, 1$)
end
AA($p$)
for $i \leftarrow 1$ to $n$ do
    LastAct =$s^i[|s^i|]$
    for $j \leftarrow 1$ to $|s^i|$ do
        if $j \neq |s^i|$ then
            AA($d^i_j$)
        end
        if $j = 1$ then
            AA($d^i_{|s^i|}$)
            AC($d^i_1, s^i[1]$,CNT($s^i[1], s^i$),1,CNT($s^i[1], s^i$))
            AC(LastAct,$d^i_1$,1,CNT(LastAct,$s^i$),0)
        end
        else
            AC($d^i_j, s^i[j]$,CNT($s^i[j], s^i$),1,
            BEF($s^i[j], j, s^i$))
            AC($s^i[j-1], d^i_j$,1,CNT($s^i[j-1], s^i$),
            AFT($s^i[j-1], j-1, s^i$))
        end
    end
    AC($p, s^i[1]$,CNT($s^i[1], s^i$),1,CNT($s^i[1], s^i$))
    AC(LastAct,$p$,1,CNT(LastAct,$s^i$),0)
end
foreach $(a_j, a_k) \in C$ do
    if P($a_j$)<P($a_k$) then
        AT($a_j, a_k$,(P($a_k$)−P($a_j$))$\cdot$CNT($a_k, s^i$)$\cdot$R($a_j, a_k$))
    end
end
```

Algorithm 1 presents the proposed methodology which is composed of two main parts:

- *modeling of mapping and scheduling*: models the mapping of each actor to a processing element and the PSOS of each processing element (see Section IV-A);
- *modeling of stage assignment*: models the synchronization of different pipeline stages (see Section IV-B).

In the following each part will be detailed and finally complexity of the proposed methodology will be discussed.

A. Modeling of mapping and scheduling

An approach similar to the one proposed in DSM technique described in Section III is used to model part of the design solution. However, differently from DSM, this part of the proposed methodology includes in the analyzed SDF graph also the description of the mapping solution. The SDF graph produced by DSM indeed does not contain a complete mapping solution, but only a part of it: all the pairs of actors, whose serialization has been forced, are implicitly mapped on the same processing element, while no information is provided about mapping of two actors already connected through a path in the original SDF graph. For this reason, an enriched SDF graph produced by DSM can correspond to more than one mapping solution; on the contrary, an enriched SDF graph produced by PAM corresponds only to one particular mapping solution. Inclusion of mapping information allows to refine analysis of the design solution: for example, it is possible to model the overhead due to context switching required for executing two tasks in sequence on the same processing element.

Scheduling and mapping solutions have to be included in the SDF graph at the same time. Inclusion of only mapping information indeed would require to force the mutual exclusion between the activation of some actors, but mutual exclusion cannot be modeled in a SDF graph. Boolean SDF graphs overcome this limit, but giving up to statical schedulability [11]. On the contrary, describing a mapping solution given a particular schedule does not require use of mutual exclusions since actors mapped on the same processing element are already forced to be activated in sequence.

The modeling of mapping and scheduling in the SDF graph is obtained by adding actors and channels which force the serial execution of each pair of consecutively activated actors of each PSOS. Given two activations of a PSOS, one of these two conditions holds:

- the second actor can be activated before the first: forcing serial execution is necessary to guarantee the correct PSOS;
- the second actor cannot be activated before the first: forcing serial execution would not be necessary to guarantee the selected schedule, but it is still added to model effects of mapping solution (e.g., context switching overhead).

The DSM technique takes into account only the former case, while this part of the proposed methodology covers also the latter. In particular, while the DSM technique considers only the decision states crossed during the execution of a PSOS (i.e., the states where more than one actor mapped on the same processing element can be activated), this part of the proposed methodology considers all the crossed states. A new actor is added (line 9) for each occurrence of an actor in PSOS to model the control of its execution like in *dataflow schedule graphs* [3], but adding the mapping information directly to the SDF graph like in [2]. Each added actor is directly connected through a channel with the actor it controls (line 17) and with the actor which precedes it in the PSOS (line 18) to model the passing of the control of the corresponding processing element between actors mapped on it. The added actor can have a null execution time or an execution time equal to context switching overhead if its effect has to be modeled in the SDF graph. If the controlled actor is the first in the schedule, the previous actor to be considered is the last in the schedule (lines 11-14).

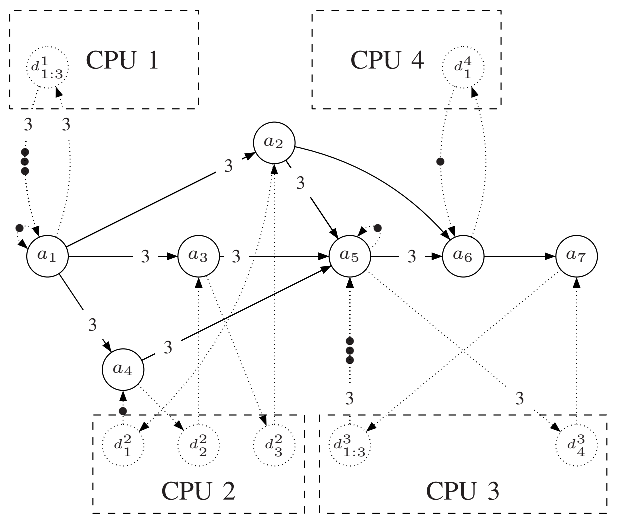

Fig. 3. SDF graph enriched with mapping and scheduling information.

Although the main difference with DSM technique is the application even in non-decision states, this part of the proposed methodology produces different results also when considering decision states where some redundant channels are not added. Since the ordering of the actor activations which compose a PSOS is absolute, this can be described by simply specifying the order between each pair of activations: all the other relative orderings will be implied by these. DSM forces the actor of choice of a decision state to be executed before all the other active actors, but to guarantee the satisfaction of the required schedule it is sufficient to impose that in each decision state the actor of choice is executed before the actor which follows it in the schedule (lines 8-19). This optimization, like the ones proposed in [2], aims at reducing the size of the produced graph, but since they are not equivalent, they have all to be applied in order to produce the smallest graph. In this way, this part of the proposed methodology adds no more than one actor and two channels for each occurrence of an actor in the PSOS. Note that all these optimizations are optional since the SDF graph produced without them would already correctly model the analyzed pipelined application, but with a larger number of actors and channels.

The last change with respect to DSM reduces the number of added self-edges by limiting them to the actors which are auto-concurrent in the given schedule (lines 1-3); these can be easily identified during analysis of decision states: these actors are activated two consecutive times in the schedule or they are opponent actors in a state reached after their activation. Note that other actors can be auto-concurrent, but the concurrent activations can occur only in states not reached in the examined schedule, so they can be ignored.

Figure 3 shows the effects of including mapping and scheduling solution of Figure 2 in the SDF graph of Figure 1. (i.e., when only this part of the methodology is applied). Note that optimizations have been applied to reduce the number of added actors and channels. $d^1_{1:3}$ and $d^3_{1:3}$, controlling multiple activations of an actor (3 activations of a_1 and 3 activations of a_5 respectively), are the result of the collapsing optimization described in [2]. The actors, which control the activation of actors mapped on the same processing element and so compose the description of the mapping solution, are enclosed in dashed boxes. Further simplifications of the produced graph have been obtained thanks to the previously presented optimizations:

- channel (d_2^2, a_2) is not added since the order to be forced (i.e., a_2 executed after a_4) is already implied by other partial orderings (i.e., a_3 after a_4 and a_2 after a_3) through added channels (i.e., (a_4, d_2^2), (d_2^2, a_3) and (a_3, d_3^2), (d_3^2, a_2));
- auto-edges (a_2, a_2), (a_3, a_3), (a_4, a_4), (a_6, a_6), (a_7, a_7) are not added since these actors are not auto-concurrent in the given schedule.

B. Modeling of stage assignment

To complete the modeling of a pipelined application, the information about stage assignment has to be integrated in the SDF graph. All the pipeline stages must be allowed to start immediately their execution during a pipeline iteration (pipeline is assumed to be fully filled), so supplementary initial tokens have to be added to all the channels which connect two actors assigned to different pipeline stages (line 26). Note that tokens have to be added even if the target is not the first actor of a pipeline stage. The number of initial tokens to be added depends on three factors:

- `CNT(`a_k, s^i`)`: how many times the target actor is activated during a pipeline iteration;
- `R(`a_k, a_j`)`: the number of tokens consumed by the actor at each activation;
- `P(`a_k`)-P(`a_j`)`: the stage distance between the source and target actors.

The first two factors have to be taken into account to allow the activation of the target actor the correct number of times. The last factor models the delay between non-consecutive stages of the pipeline and so the requirement of larger buffer for these channels. Added tokens can create new decision states, but since the previous step of the proposed methodology already forces the serialization in non-decision states, no further modifications are required to guarantee the schedule solution. For example a_5 and a_7 could now be executed at the same time, but their serialization was already forced by d_4^3.

Finally, the synchronization among different stages of the pipeline has to be forced: a new iteration may not start before the ending of execution of all the pipeline stages. For this reason, an actor p (line 4), whose activation identifies the starting of a new pipeline iteration, is introduced. This actor should be connected with all the actors of the application (with the opportune initial tokens) and all the actors should be connected with it so that all the actors would be activated the correct number of times before the starting of a new iteration. To reduce the size of the graph, these channels can be added in a selective way: actor p is connected only to the first actor of each PSOS (line 21). Since the activations of the actors assigned to the same processing element have already been sequentialized, adding these channels guarantee that all the actors of the SDF graph have to wait the activation of p in order to be activated. In the same way, channels incoming into

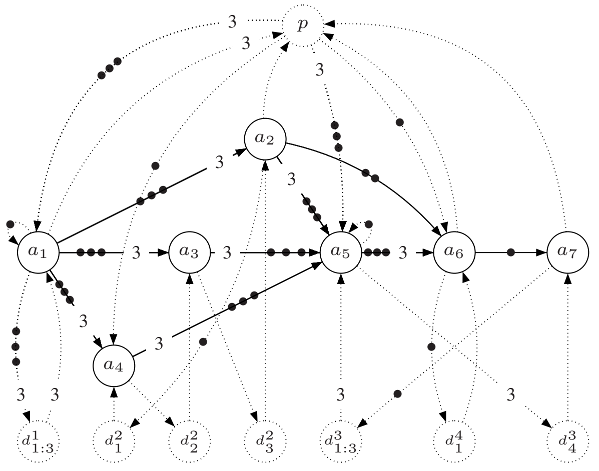

Fig. 4. SDF graph enriched with complete solution information.

p are added only from all the last scheduled actors (line 22): this guarantees that p can be activated only after the execution of all the actors of the SDF graph. On the contrary, adding only the channel from p to the first actor of the first stage and from the last actor of the last stage to p is not enough. In this case, activation of all the actors of the graph would be guaranteed, but not the synchronization of the different pipeline stages.

Figure 4 shows the SDF graph describing the design solution of Figure 2 obtained by applying Algorithm 1. Supplementary tokens have been added to all the channels of the original SDF graph since all pairs of actors connected by a channel are assigned to different pipeline stages. Note that two tokens instead of one have been added to channel (a^2, a^6) to correctly model the delay between non-consecutive stages of the pipeline. The actor p, which controls the synchronization of the pipeline stages, has been added and it has been connected with all the first scheduled actors (i.e., a_1, a_4, a_5, a_6) and all the last scheduled actors (i.e., a_1, a_2, a_6, a_7) have been connected to it. Note that it is not necessary to connect p to a_7 to synchronize the starting of the stage 4 of the pipeline, nor it is necessary to connect a_5 to p to synchronize the ending of stage 2. Since these two stages are mapped on the same processing element, they are indeed implicitly sequentialized.

C. Complexity of the Proposed Methodology

The complexity of PAM is $\Theta(\sum_{i=0}^{n} |s^i| + |C|)$: the first term is given by the loop body in lines 8-19 which is executed $\sum_{i=0}^{n} |s^i|$ times, the second term is given by the loop in lines 24-28. The number of added actors is $\sum_{i=0}^{n} |s^i|+1$, but it can be reduced to $\sum_{i=0}^{n} App(s^i)+1$ (where $App(s^i)$ sums the appearances of each actor in PSOS s^i). The maximum number of added channels is $|A_a| + 2 \cdot \sum_{i=0}^{n} App(s^i) + 2 \cdot n$, where A_a is the set of auto-concurrent actors and n is the number of processing elements.

V. Case Study

In this section an application scenario of the proposed methodology is presented: evaluation of pipelined application design solutions during design space exploration. This can

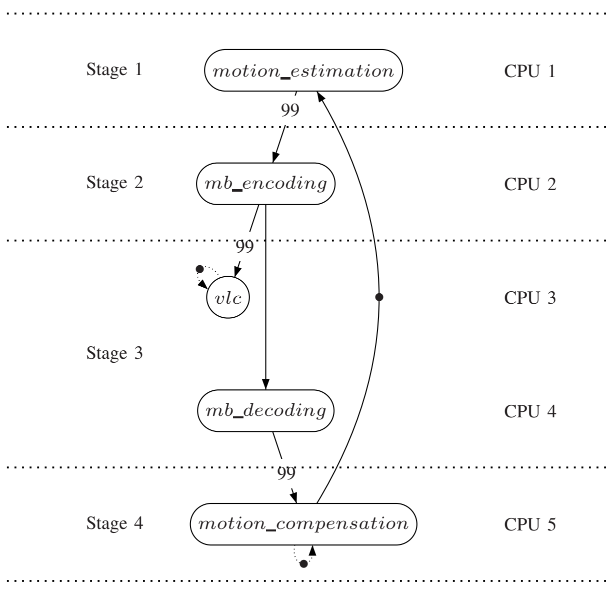

Fig. 5. SDF graph representing H263 decoder and pipelined solution to be represented.

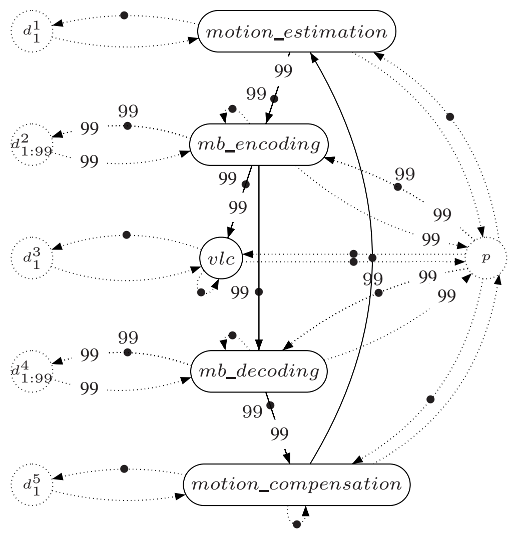

Fig. 6. SDF graph produced by PAM starting from SDF graph of Fig. 5.

TABLE I. RESULTS OF APPLICATION OF PROPOSED METHODOLOGY.

		Starting SDF		Produced SDF	
Benchmark	DS Size	|A|	|C|	|A|	|C|
modem	$2.09 \cdot 10^{13}$	16	36	33	68
sample-rate converter	$7.20 \cdot 10^{2}$	6	11	13	23
satellite receiver	$1.12 \cdot 10^{21}$	22	48	45	92
mp3 decoder	$871 \cdot 10^{10}$	14	19	25	58
mp3playback	$2.40 \cdot 10^{1}$	4	8	9	16
H263 decoder	$2.40 \cdot 10^{2}$	4	6	9	22
H263 encoder	$1.20 \cdot 10^{2}$	5	7	11	19

be easily obtained by combining a design space exploration technique, the methodology proposed in this paper and a tool for the analysis of SDF graphs like SDF^3 [5]. In the design of pipelined applications, design space exploration techniques are usually required since the design solution space, even for relatively small applications, can be very huge because of the several elements which compose each solution (i.e., the mapping and the pipeline stage assignment of each actor, the schedule on each processing element). Left part of Table I reports the size of the design solution space of stage assignment problem for some common SDF benchmarks (modem [12], sample-rate converter [12], satellite receiver [13], mp3playback [14], H263 decoder [15], H263 encoder [16] and MP3 decoder [15]). Note that, even considering only the pipeline stage assignment problem, the solution design space can be quite large even on simple applications. For this reason, use of design space exploration techniques coupled with high abstraction level representations is mandatory to reduce the number of solutions to be analyzed during the design process and to reduce the complexity of the analysis of each solution. In the following, no particular design space exploration technique will be considered: the presented results do not depend on it and evaluating a particular design space exploration technique is out of the scope of this paper.

To be applied to solve a design problem, a design space exploration technique usually requires: one or more figures of merit, to quantify the goodness of a solution, and a method to compute or estimate these quantities for each analyzed solution. The two typical metrics adopted to evaluate a pipelined solution are the maximum throughput [15] and the minimum buffer size which guarantees it [17]. Both these quantities can be easily computed analyzing a SDF graph, for example by exploiting SDF^3 [5], but these analyses have to take into account mapping, scheduling and pipeline stage assignment in order to produce estimations enough accurate to correctly evaluate a design solution.

The analysis of the throughput of the H263 encoder benchmark, whose SDF graph is shown in Fig. 5, highlights the errors which can be introduced in estimating pipelined applications. Suppose that the design solution to be evaluated is the one presented in Fig. 5: each actor has been mapped on a different processing element; vlc and $mb\_decoding$ have been assigned to the same pipeline stage, while $motion\_estimation$, $mb\_encoding$ and $motion\_compensation$ have been each assigned to a different pipeline stage. The PSOSs of the analyzed solution are: $s^1 = (motion\_estimation)$, $s^2 = (mb\_encoding^{99})$, $s^3 = (vlc)$, $s^4 = (mb\_decoding^{99})$, $s^5 = (motion\_compensation)$. The throughput computed with $SDF3^3$ considering the SDF graph produced by DSM technique is 20.28 frames per second while the computed overall buffer size is 2.04 MB. According to this SDF graph, each actor has to wait that all the others actors fire again in sequence before being activated again, with the exception of $mb\_encoding$ and $mb\_decoding$ which executions can be overlapped.

The proposed methodology has been applied to include the pipelined solution described in Fig. 5 in the SDF graph of the H263 encoder. The obtained SDF graph, which is shown in Fig. 6, has been written as an XML file compliant to the SDF^3 XML syntax: in this way it has been possible to compute its throughput and the minimum buffer sizes which guarantee it by means of SDF^3 without reimplementing any analysis techniques. The obtained throughput is almost 1.5x better than the one computed ignoring pipeline assignment information (30,03 frames per second) while overall buffer size is 1.5x worse (3.02MB). Indeed, the proposed methodology adds supplementary initial tokens to all the edges of the produced SDF graph to correctly model that the activations of all the actors are overlapped. For this reason, the throughput of the whole application is determined by the slowest pipeline stage of the analyzed solution that is the one composed of actor $motion\_estimation$. At the same time, because of the added initial tokens, the computed overall buffer size is larger.

After having shown on a real application case study how the proposed methodology can be easily integrated with existing framework and how much the added information can impact on analysis results, some data about the runtime of the methodology are reported. Indeed, in order to be suitable to be integrated in a design space exploration methodology, an evaluation technique must be fast: the faster it is, the larger the number of evaluations that can be performed in a same amount of time. In the considered scenario, the evaluation technique is composed of the proposed methodology and of the analyses performed by SDF^3. The time complexity and so the execution time of PAM depend on the size of the initial SDF graph while the execution time of the analyses performed by SDF^3 depends on the size of the SDF graph produced by PAM. Details about the size of the analyzed applications and about the size of the enriched graphs are reported in right part of Table I. The size in terms of numbers of actors and channels of the produced SDF graphs refers to single-appearance solutions. Note that, as described in Section IV, PAM can be applied to this type of solutions, which are usually adopted in the design of streaming applications, but also to non single-appearance solutions. Since throughput and buffer size analyses integrated in the SDF^3 are very fast techniques [18], their precise execution times cannot be collected because of limited resolution of profiling mechanisms available on the host machine. The analyses require for each one of the considered enriched graphs less than 10 milliseconds; a non-optimized C++ implementation of PAM requires less than 100 milliseconds to produce an enriched graph, but thanks to the small complexity of the proposed methodology, much better performances can be obtained with an efficient implementation. As a result, the combination of the proposed methodology and of SDF graph analyses can be considered enough fast to be fully suitable for exploitation in a fast design space exploration methodology.

VI. Conclusions

In this paper a methodology to describe all the aspects of a pipelined application by means of a SDF graph have been presented. The methodology is composed of two main parts: the former adds mapping and scheduling information, the latter stage assignment information. The usage of SDF graphs facilitates the portability of the design solutions allowing the application of existing techniques for analysis and synthesis of embedded systems based on SDF graphs and reduces the gap between a high level description of the design solution and its implementation.

References

[1] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235 – 1245, 1987.

[2] M. Damavandpeyma, S. Stuijk, T. Basten, M. Geilen, and H. Corporaal, "Modeling static-order schedules in synchronous dataflow graphs," in *DATE '12*, 2012, pp. 775–780.

[3] H.-H. Wu, C.-C. Shen, N. Sane, W. Plishker, and S. S. Bhattacharyya, "A model-based schedule representation for heterogeneous mapping of dataflow graphs," in *IPDPSW '11*, 2011, pp. 70–81. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2011.128

[4] Y. Chen and H. Zhou, "Buffer minimization in pipelined sdf scheduling on multi-core platforms," in *ASP-DAC '12*, 2012, pp. 127–132.

[5] S. Stuijk, M. Geilen, and T. Basten, "Sdf3: Sdf for free," in *ACSD '06*, 2006, pp. 276–278.

[6] A. Navarro, R. Asenjo, S. Tabik, and C. Cascaval, "Analytical modeling of pipeline parallelism," in *PACT '09*, 2009, pp. 281 –290.

[7] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers*, vol. 36, no. 1, pp. 24–35, 1987. [Online]. Available: http://dx.doi.org/10.1109/TC.1987.5009446

[8] N. K. Bambha, V. Kianzad, M. Khandelia, and S. S. Bhattacharyya, "Intermediate representations for design automation of multiprocessor dsp systems," *Design Automation for Embedded Systems*, vol. 7, no. 4, pp. 307–323, 2002.

[9] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "Streamit: A language for streaming applications," in *CC '02*, 2002, pp. 179–196. [Online]. Available: http://dl.acm.org/citation.cfm?id=647478.727935

[10] A. Pop and A. Cohen, "A stream-computing extension to openmp," in *HiPEAC '11*, 2011, pp. 5–14. [Online]. Available: http://doi.acm.org/10.1145/1944862.1944867

[11] S. S. Bhattacharyya, E. F. Deprettere, and J. Keinert, "Dynamic and multidimensional dataflow graphs," in *Handbook of Signal Processing Systems*, S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, Eds., 2010, pp. 899–930.

[12] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, "Synthesis of embedded software from synchronous dataflow specifications," *Journal of VLSI Signal Processing Systems*, vol. 21, no. 2, pp. 151–166, 1999. [Online]. Available: http://dx.doi.org/10.1023/A:1008052406396

[13] S. Ritz, M. Willems, and H. Meyr, "Scheduling for optimum data memory compaction in block diagram oriented software synthesis," in *ICASSP '95*, vol. 4, 1995, pp. 2651–2654 vol.4.

[14] M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit, "Efficient computation of buffer capacities for cyclo-static dataflow graphs," in *DAC '07*, 2007, pp. 658–663. [Online]. Available: http://doi.acm.org/10.1145/1278480.1278647

[15] A. H. Ghamarian, M. Geilen, S. Stuijk, T. Basten, B. D. Theelen, M. R. Mousavi, A. J. M. Moonen, and M. Bekooij, "Throughput analysis of synchronous data flow graphs," in *ACSD '06*, 2006, pp. 25–36.

[16] H. Oh and S. Ha, "Fractional rate dataflow model for efficient code synthesis," *J. VLSI Signal Process. Syst.*, vol. 37, no. 1, pp. 41–51, 2004. [Online]. Available: http://dx.doi.org/10.1023/B:VLSI.0000017002.91721.0e

[17] J. Park and W. J. Dally, "Buffer-space efficient and deadlock-free scheduling of stream applications on multi-core architectures," in *SPAA '10*, 2010, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/1810479.1810481

[18] A.-H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, A. J. M. Moonen, M. Bekooij, B. Theelen, and M. Mousavi, "Throughput analysis of synchronous data flow graphs," in *ACSD '06*, 2006, pp. 25–36.