SPDF: A Schedulable Parametric Data-Flow MoC

Pascal Fradet[†] Alain Girault[†] Peter Poplavko[†,‡]
INRIA Grenoble Rhône-Alpes[†] CRI-PILSI[‡]
first.last@inria.fr

***Abstract*—Dataflow programming models are suitable to express multi-core streaming applications. The design of high-quality embedded systems in that context requires static analysis to ensure the liveness and bounded memory of the application. However, many streaming applications have a dynamic behavior. The previously proposed dataflow models for dynamic applications do not provide any static guarantees or only in exchange of significant restrictions in expressive power or automation. To overcome these restrictions, we propose the *schedulable parametric dataflow (SPDF)* model. We present static analyses and a quasi-static scheduling algorithm. We demonstrate our approach using a video decoder case study.**

I. Introduction

Multi-core systems are becoming an increasingly important platform for many embedded system designs. To take advantage of multi-cores, programming languages should express thread-level parallelism. Among such languages, *dataflow* languages are prominent for many streaming applications [1].

Recent dataflow programming environments support applications whose behavior is characterized by dynamic variations in resource requirements. The high expressive power of the underlying models makes it challenging to ensure predictable behavior. For example, the CAL actor language [1] or Kahn Process Networks [2] can express many dynamic applications. However, checking *liveness* (*i.e.,* no part of the system will deadlock) and *boundedness* (*i.e.,* can be executed in finite memory) is known to be hard or even undecidable.

This situation is troublesome for the design of high-quality embedded systems. Sufficient criteria for liveness and boundedness have been formulated for less expressive models, which can nevertheless express the core part of many streaming applications. However, such *statically analyzable* criteria come at the cost of significantly constraining modeling and scheduling. For example, parametrical synchronous dataflow (PSDF) [3] imposes a hierarchical discipline which restricts scheduling and analysis.

In this paper, we introduce the *schedulable parametric dataflow (SPDF)* model of computation (MoC) for dynamic streaming applications. SPDF was designed to be statically analyzable for liveness and boundedness, while avoiding the aforementioned restrictions. In Section II, we present a well-known basic model – synchronous dataflow (SDF) [4] – which is easily analyzable but restricted to static applications. We then introduce our SPDF model as a parametric and dynamic extension of SDF. In Section III, we present the static analyses for boundedness and liveness. Section IV describes the compilation, such as insertion of parameter distribution network and quasi-static scheduling. A video decoding application is presented as a case study in Section V. Finally, Section VI summarizes our contribution, compares it to related work and hints at future research directions.

II. Model of Computation

We start from SDF – synchronous dataflow [4] – one of the simplest dataflow MoC. Then, we present our MoC (SPDF) as a statically analyzable extension of SDF with dynamic parametrization.

A. Basic Model: SDF

In SDF, a program is defined by a directed graph, where nodes – called *actors* – are functional units. The actors have *data ports* connected by *edges* which can be seen as FIFO (first-in first-out) channels. The atomic execution of a given actor – called *actor firing* – consumes data tokens from its incoming edges (its *inputs*) and produces data tokens to its outgoing edges (its *outputs*). The number of tokens consumed or produced at a given port at each firing is called the *rate*. It is denoted as $r(\pi_m)$ where π_m is a port. In SDF, all rates are constant and known at compile time.

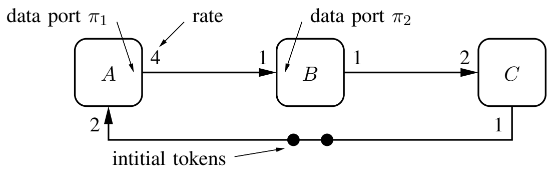

Fig. 1. A simple SDF graph.

Fig. 1 shows a simple SDF graph with three interconnected actors A, B and C. Actor A has one input and one output port, whose rates are 2 and 4, respectively.

The *state* of a dataflow graph is the number of tokens present at each edge (i.e., buffered in each FIFO). Each edge carries zero or more tokens at any moment of time. The initial state of the graph is specified by the number of *initial tokens*. Edge (C, A) in Fig. 1 has two initial tokens. After the first firing of actor A, the edge (A, B) gets four tokens while the two tokens of (C, A) are consumed.

A major advantage of SDF is that, if it exists, a bounded schedule can be found statically. Such a schedule ensures that each actor is eventually fired (ensuring liveness) and that the graph returns to its its initial state after a certain sequence of firings (ensuring boundedness of the FIFOs). The minimal such sequence is called an *iteration*.

The numbers of firings of the different actors per iteration are computed by solving the so-called *system of balance equations*. This system is made of one equation per edge. Consider an edge (X_1, X_2) connecting the ports π_1 and π_2; its balance equation is:

$$\#X_1 \cdot r(\pi_1) = \#X_2 \cdot r(\pi_2) \tag{1}$$

This equation states that the number of firings of the producer X_1, denoted $\#X_1$, multiplied by its rate $r(\pi_1)$, should be equal to the same expression for the consumer X_2. For example, the balance equation for edge (A, B) in Fig. 1 is: $\#A \cdot 4 = \#B \cdot 1$.

The existence of solutions of the system of balance equations is referred to as *rate consistency*. The graph of Fig. 1 is rate-consistent, and the solutions are: $\#A = 1$, $\#B = 4$ and $\#C = 2$. Note that multiplying the solutions by the same positive constant makes another set of solutions. One usually considers only the minimal strictly positive integer solutions which are obtained by eliminating common factors.

The minimal solutions determine the number of firings of each actor per iteration. The next step is to determine a static order – the *schedule* – in which those firings can be executed. The schedule is obtained by an abstract computation where an actor is fired only when it has enough input tokens. The graph of Fig. 1 can only start by firing A; then, B has enough input tokens to be fired four times, and finally C twice. Since each actor has been fired the exact number of times requested by its solution, a schedule has been found. We represent it as the string AB^4C^2 where the superscripts denote repetition count. Another valid schedule for the same graph is AB^2CB^2C which can also be written as $A(B^2C)^2$.

B. Our model: SPDF

We extend SDF by allowing rates to be parametric while preserving static schedulability. Let $\mathcal{P}$ be a set of symbolic variables. SPDF rates are defined by the grammar:

$$\mathcal{F} \quad ::= \quad k \mid p \mid \mathcal{F}_1 \cdot \mathcal{F}_2 \qquad \text{where } k \in \mathbb{N}^* \text{ and } p \in \mathcal{P}$$

Rates are products of positive integers (k) or symbolic variables (p). Optionally, each parameter can be constrained to belong to a specific integer interval ($[1, +\infty)$ by default).

Fig. 2 shows a simple SPDF graph where the actors have constant or parametric rates (e.g., $p \cdot q$ for the input rate of C).

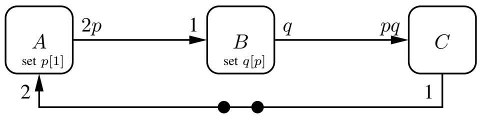

Fig. 2. A simple SPDF graph

Unlike the rates of SDF graphs which are fixed at compile time, the parametric rates of a SPDF graph can change dynamically. The changes of each parameter are made by a single actor called its *modifier*. By default, a parameter can be changed between iterations. In SPDF, a modifier may change a parameter more often using the annotation "*set* $p[\alpha]$" where p is the parameter to be set and α is the exact (possibly symbolic) number of firings of the modifier between two parameter changes. In Fig. 2, A and B are the modifiers for p and q; they may change their value every single and p firings, respectively.

Definition 1: A SPDF graph is a tuple $(\mathcal{G}, \mathcal{P}, i, d, r, M, \alpha)$, where:

- $\mathcal{G}$ is a directed connected graph $(\mathcal{A}, \mathcal{E})$ with $\mathcal{A}$ a set of actors and $\mathcal{E} \subseteq \mathcal{A} \times \mathcal{A}$ a set of directed edges;
- $\mathcal{P}$ is a set of parameters;
- $i : \mathcal{E} \to \mathbb{N}$ associates each edge with its number of initial tokens;
- $d : \mathcal{P} \to 2^{\mathbb{N}^*}$ returns the interval of each parameter;
- $r : \mathcal{A} \times \mathcal{E} \to \mathcal{F}$ returns for each port (represented by an actor and one of its edges) its associated (possibly symbolic) rate;
- $M : \mathcal{P} \to \mathcal{A}$ and $\alpha : \mathcal{P} \to \mathcal{F}$ return for each parameter its modifier and its change period, respectively.

III. Static Analysis for SPDF

This section presents the three static analyses needed to ensure boundedness and liveness of an SPDF graph. In Section III-A, we check rate consistency by adapting the analysis of SDF to SPDF. Conditions for consistency and solutions of balance equations are computed in terms of symbolic expressions. In Section III-B, we check that the change periods of each parameter are safe. Rate consistency and parameter change safety ensure boundedness. Section III-C completes the analysis chain by checking for liveness.

A. Rate Consistency

As in SDF, we check the rate consistency of a SPDF graph by generating the associated system of balance equations. This system must be shown to have a solution for all possible values of parameters. When parameters are modified only between iterations, rate consistency ensures boundedness.

We generalize the algorithm for solving the balance equations in SDF presented in [5] to SPDF by doing the same operations with symbolic factors. That algorithm relies on multiplication, division and greatest common divisor (*gcd*) of rates. These operations are easily expressed on $\mathcal{F}$ by putting symbolic expressions on the form:

$$\underbrace{k_0 \cdot k_1 \cdot k_2 \cdots}_{\text{prime decomposition}} \cdot \underbrace{p_1 \cdot p_1 \cdots p_1}_{\text{the power of } p_1} \cdot \underbrace{p_2 \cdot p_2 \cdots p_2}_{\text{the power of } p_2} \cdots$$

The minimal solutions for all actors are found by eliminating all the prime or parameter factors common to all solutions.

If the undirected version of the SPDF graph is acyclic, a solution always exists and will be found by the above computation. When the SPDF graph contains an undirected cycle, the graph may be rate inconsistent. There is, however, a necessary and sufficient condition for the existence of solutions. Each undirected cycle $X_1, X_2, \ldots, X_n, X_1$ where p_i and q_j denote the rates of edge (X_i, X_j) should satisfy the following condition:

$$(\textit{Cycle condition}) \quad p_1 \cdot p_2 \ldots \cdot p_n = q_1 \cdot q_2 \ldots \cdot q_n$$

This condition enforces that any factor encountered on an "output" port of a cycle should have a *symbolically identical* counterpart on an "input" port on this cycle.

Theorem 2 (Consistency): An SPDF graph is rate consistent if its undirected cycles satisfy the cycle condition [6].

For example, the graph of Fig. 2 is consistent since its only cycle A, B, C, A satisfies the cycle condition which is $2p \cdot q \cdot 1 = 2 \cdot 1 \cdot pq$. The minimal solutions are $\#A = 1$, $\#B = 2 \cdot p$ and $\#C = 2$, yielding the schedule $AB^{2p}C^2$.

The specified algorithm either yields for each actor its (symbolic) solution, or returns an unsatisfied cycle condition that can be used by the programmer to fix his SPDF graph.

B. Parameter Change Safety

It is always safe to change parameter values between graph iterations [3]. Indeed, the rate consistency and liveness analyses ensure that the graph is bounded and live for any value of the parameters. Since the graph returns to its initial state after each iteration, all parameters can be modified at these stages. Nevertheless, it is sometimes useful to change the parameters more often, i.e., during an iteration. SPDF allows the programmer to specify a faster period using the "*set* $p[\alpha]$" annotation. Yet, not all periods are safe and their consistency must be checked. Consider, for instance, actor B that modifies q in Fig. 2. The period 1 would not be safe since it is only after p firing of B that C can consume its pq tokens. The rate pq would not be well defined if q can change p times before C is fired. On the other hand, the period p is safe since the iteration can be written $A(B^pC)(B^pC)$, with q being changed after each sequence (B^pC).

The criterion ensuring that parameter modification periods are safe relies on the notions of *influence*, *regions* and *local iterations*. Intuitively, the criterion states that a parameter can be modified once per local iteration of the region it influences. For Fig. 2, it can be shown that the region of influence of q consists of actors B and C and that q can be changed after each local iteration (B^pC), that is, after p firings of B.

Definition 3 (Influence): An edge $e = (A, B)$ is *influenced* by a parameter p, denoted $\mathit{Infl}(e, p)$, if p appears in the rates of e or in the solutions of the balance equations of its source and sink actors. Formally,
$\mathit{Infl}(e, p) \Leftrightarrow p \in \#A \vee p \in \#B \vee p \in r(A, e) \vee p \in r(B, e)$
where $p \in \mathcal{F}$ if p occurs in the symbolic expression $\mathcal{F}$.

The *region* of influence of a parameter is the subset of edges it influences. Since an edge is a relation between actors, a region also specifies a subset of actors.

Definition 4 (Region): The region of edges $\mathcal{R}(p)$ influenced by p is defined as: $\mathcal{R}(p) = \{e \mid \mathit{Infl}(e, p)\}$

We will sometimes abuse notation $\mathcal{R}$ to denote also the set of actors connected by the edges of the region. For example, the region of influence of q in Fig. 2 is $\mathcal{R}(q) = \{(B, C)\}$ and the actors in this region are $\{B, C\}$.

The solutions of the system of balance equations are *global solutions* in that they define the number of firings for the global iteration of the whole graph. Local solutions are solutions for a subset of actors; they denote a nested iteration.

Definition 5 (Local solutions): Let $\mathcal{A}$ be the set of actors of an SPDF graph and $\#X$ be the global solution of X. The local solution of X in the subset $\{X_1, \ldots, X_n\} \subset \mathcal{A}$, denoted $\#_{\mathrm{L}}X$, is obtained by dividing the global solution of X by the greatest common divisor: $\#_{\mathrm{L}}X_i = \frac{\#X_i}{gcd(\#X_1, \ldots, \#X_n)}$.

For example, the global solutions for Fig. 2 are $\#A = 1$, $\#B = 2p$ and $\#C = 2$, forming the global iteration $AB^{2p}C^2$. The *gcd* of $\#B$ and $\#C$ is 2 and the local solutions for the subset $\{B, C\}$ are $\#_{\mathrm{L}}B = p$ and $\#_{\mathrm{L}}C = 1$. After one local iteration B^pC, all the edges influenced by q return to their initial state. Therefore, q can be changed after each such local iteration, hence after p firings of B, as specified by the "*set* $q[p]$" annotation.

Regions of influence of a given parameter can overlap (i.e., have common edges). Each local iteration of such region may entail firing the same actor a different number times. Such overlapping regions must be grouped so that the modification periods of their parameters are checked on the same subset of actors. Regions are then generalized to a subset of parameters $\mathcal{P}'$ as follows:

$$\mathcal{R}(\mathcal{P}') = \{e \mid \exists p \in \mathcal{P}', \mathit{Infl}(e, p)\}$$

When a region $\mathcal{R}(\mathcal{P}_2)$ is included within another region $\mathcal{R}(\mathcal{P}_1)$, the periods of the parameters in $\mathcal{P}_2$ can be checked on $\mathcal{R}(\mathcal{P}_2)$. The local iteration of $\mathcal{R}(\mathcal{P}_1)$ will always involve one or several local iterations of the inner region. Hence, the changes of parameters from $\mathcal{P}_1$ are always done between local iterations of $\mathcal{R}(\mathcal{P}_2)$ and are therefore safe for both regions.

Before checking the parameter modification safety criterion, we structure the set $\mathcal{P}$ of all parameters into a *hierarchy tree* of sets of parameters $\mathcal{P}_i$ such that:

- $\mathcal{P}$ is partitioned into non-empty partitions $\mathcal{P}_i$ that are placed at different nodes and leafs of the hierarchy;
- the region of a set $\mathcal{P}_i$ is strictly included in the region of its parent $\mathcal{P}_j$ (i.e., $\mathcal{R}(\mathcal{P}_i) \subset \mathcal{R}(\mathcal{P}_j)$);
- the regions of two sets $\mathcal{P}_i$ and $\mathcal{P}_j$ which are not ancestor or descendant of each other are disjoints.

This structuring process is based on two basic steps:

- (*Decomposition*) the first step decomposes the current set of edges (initially $\mathcal{E}$) into disjoint regions. Consider the relation $e_1 \asymp e_2$ which holds if there exists a parameter influencing both edges e_1 and e_2. Then, disjoint regions are the connected components of the graph of the $\asymp$ relation. Each disjoint set of edges corresponds to a region of a disjoint set of parameters;
- (*Nesting*) the second step finds, for each such independent region $\mathcal{R}(\mathcal{P})$, the largest subset $\mathcal{P}' \subset \mathcal{P}$ such that $\mathcal{R}(\mathcal{P}') \subset \mathcal{R}(\mathcal{P} - \mathcal{P}')$. The set $\mathcal{P} - \mathcal{P}'$ will be the root of the (sub-)tree that will be built by iterating the process (decomposition and nesting) on $\mathcal{P}'$. This process ends when $\mathcal{P}' = \emptyset$.

Fig. 3 represents a graph with two hierarchy levels: the parent level $\mathcal{P}_1 = \{p\}$ and the child level $\mathcal{P}_2 = \{q\}$. The parameter p influences all edges whereas q influences only (A, D) and (D, C), hence $\mathcal{R}(\mathcal{P}_2) \subset \mathcal{R}(\mathcal{P}_1)$. We can now state the criterion for parameter modification safety.

Definition 6 (Data Safety): An SPDF graph is *data safe* if, for each parameter p and its hierarchy node $\mathcal{P}_i$ such that $p \in \mathcal{P}_i$, every actor X_j in $\mathcal{R}(\mathcal{P}_i)$ is such that $\#X_j$ is a multiple of $\#M(p)/\alpha(p)$.

This criterion ensures that when a parameter takes a new value between two local iterations of its hierarchy level, all the data edges come back to their initial state. In Fig. 3, we have $q \in \mathcal{P}_2$, $\#M/\alpha = \#A/1 = 2$, and $\mathcal{R}(\mathcal{P}_2) = \{A, C, D\}$. The solutions for the actors in $\mathcal{R}(\mathcal{P}_2)$ are all multiples of 2: $\#A = 2$, $\#C = 2p$ and $\#D = 2pq$. The annotation "*set* $q[1]$" in A is thus data safe.

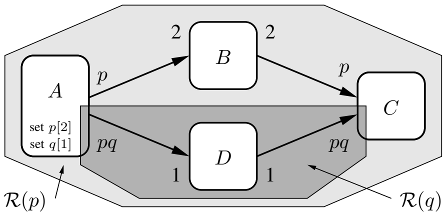

Fig. 3. An SPDF graph with two hierarchy levels: $\mathcal{P}_1 = \{p\}$, $\mathcal{P}_2 = \{q\}$.

Definition 7 (Period Safety): An SPDF graph is *period safe* if, for every pair of parameters p and q such that $\#M(q)$ depends on at least one parameter of the hierarchy node of p, $\#M(q)/\alpha(q)$ is a multiple of $\#M(p)/\alpha(p)$.

This criterion ensures that every modifier is contained in at least one region whose local iterations never finish when a period of that modifier is not yet completed. E.g., the graph of Fig. 2 is period-safe because even if the solution of $M(q)$ depends on p ($\#M(q) = \#B = 2p$), $\#B/\alpha(q) = 2$ is a multiple of $\#M(p)/\alpha(p) = 1$, so the criterion is satisfied.

Theorem 8 (Boundedness): If an SPDF graphs is rate consistent, data safe and period safe, then all data edges and periods come back to the initial state at the end of a global iteration. Hence it can be scheduled in bounded memory.

C. Liveness

If the directed SPDF graph is acyclic it is trivially live: there exist actors that can always fire, thus allowing other actors to fire, and so on until the iteration is complete. However, if there exists a directed cycle, we must check that each cycle contains enough initial tokens. For example, if the (C, A) edge in Fig. 2 had only one token, then A (and therefore B and C) could never fire. Checking the liveness of SDF graphs is done by computing an iteration by abstract execution. It is not clear whether such an approach is applicable to SPDF. Instead, we present a sufficient condition on cycles.

Definition 9 (Live cycle): Consider a cycle consisting of n actors X_1, X_2, ... X_n and n edges $e_1 = (X_n, X_1)$, $e_2 = (X_1, X_2)$, ... $e_n = (X_{n-1}, X_n)$. We say that the cycle is *live* if $\exists 1 \leq k \leq n$ such that $i(e_k) \geq r(X_k, e_k) \cdot \#X_k$, i.e., there is an edge with enough initial tokens to fire an actor the needed number of times to complete the iteration.

If $r(X_k, e_k) \cdot \#X_k$ is a symbolic expression, the inequality is checked using the maximum values of the parameters involved. If one of the parameters does not have a declared maximum, then the inequality is considered false. In Fig. 2, the cycle is live since $i(C, A) = 2$, $r(A, (C, A)) = 2$, and $\#A = 1$.

Definition 10 (User): A *user* U of a parameter p is an actor different from $M(p)$ such that p occurs in $\#U$ or in the rate of one of the ports of U.

Theorem 11 (Liveness): A rate-consistent and safe SPDF graph is live if each of its cycles is live and if, for each parameter, there is path from the modifier to all the users without initial tokens [6].

The second requirement ensures that the parameter communication from the modifier to the users does not introduce non-live cycles. Liveness analysis either succeeds or returns to the programmer the faulty cycles (i.e., with not enough initial tokens) or the faulty modifier-user pairs.

We could use, as in [3] [5], less restrictive criteria using local solutions in strongly connected subgraphs. We skip this possibility here for space and simplicity reasons.

IV. Compilation

We first show how to transform any safe SPDF graph into a graph which can be scheduled in bounded memory by dynamic scheduling (Section IV-A). Then, we describe how to generate a quasi-static schedule (Section IV-B).

A. Parameter Communication for Bounded Scheduling

The critical aspect for scheduling SPDF graphs is the communication of the values of parameters from the modifiers to the users. We implement this communication by adding extra actors, edges, and ports, forming a *parameter distribution network* (PDN), in such a way that the SPDF graph remains rate consistent and safe. The PDN joins $M(p)$ to all the users of p. It is inserted in three steps.

The first step adds to $M(p)$ a new *output* port, and to each user a new *control* port, respectively to send and to receive the successive values of p. Control ports behave exactly as in BDF [7]: each actor must read input tokens from all its control ports before reading tokens from its regular data ports.

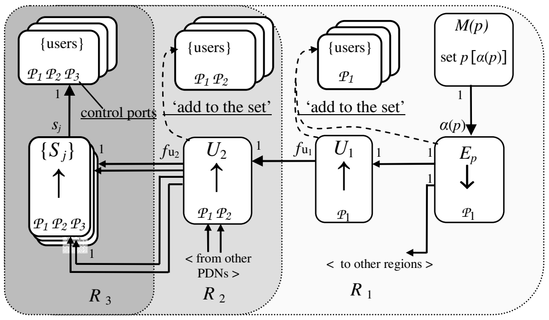

Fig. 4. Communication of a parameter from the modifier to the users.

The second step adds a new actor E_p, called the *emitter* of p, and a new edge $e = (M(p), E_p)$, such that $r(M(p), e) = 1$ and $r(E_p, e) = \alpha(p)$. Among the $\alpha(p)$ tokens it receives, E_p transmits only the first one, the one that contains the new value of p. We refer to E_p as a *downsampler* ($\downarrow$). This is illustrated in Fig. 4.

In general, E_p fires once per certain number of firings of the users, so each user should receive the same value of p repeated a certain number of times. The third step implements this requirement using *upsamplers* ($\uparrow$) that repeat every input token a given number of times. This step depends on the region hierarchy.

Definition 12 (User Location): A user X of p is said to be *located* w.r.t. *to* p at the lowest hierarchy level that contains p or a parameter that occurs in $\#X$.

Definition 13 (Edge Location): An edge is said to be *located* in the hierarchically lowest region that contains it.

The order in which the users are connected to the emitter is defined by their hierarchal location. For parameter p contained in hierarchy node $\mathcal{P}_k$ the users can be located in the node $\mathcal{P}_k$ itself or lower in the hierarchy (see Fig. 4).

It can be shown that the output edge of p's emitter is located in a higher region than the location of any user of p. Let R_1 be the region of the p's emitter output and R_N be one of the regions where p is used; with the hierarchy path: R_1, R_2 ... R_N (see Fig. 4, where $N = 3$).

Definition 14 (Local iteration w.r.t. *a larger region):* The local iteration count of a region R' in the context of a larger region R'', $R' \subset R''$, is defined as $\frac{gcd\{\#X | X \in R'\}}{gcd\{\#X | X \in R''\}}$. This local iteration count gives the number of iterations of R' per each iteration of R''.

Let us calculate the upsampling factor f_{U} that is common for all users in R_N. Let f_{R} be the number of local iterations of R_N *w.r.t.* R_1 and f_{E} be the local solution for E, definition 6 implies that f_{R} is a multiple of f_{E} and we write $f_{\mathrm{U}} = f_{\mathrm{R}}/f_{\mathrm{E}}$.

In general, f_{U} may include parameters from different regions. So, to preserve the parameter safety, instead of one upsampling actor we need an *actor chain* $(U_k)_{k=1 \ldots N-1}$ with one actor per region (see Fig. 4). Their upsampling rates are obtained by considering f_{U} as : $f_{\mathrm{U}} = f_{\mathrm{u}1} \cdot f_{\mathrm{u}2} \cdots f_{\mathrm{u}(N-1)}$.

One can show that $f_{\mathrm{u}k} = f_{\mathrm{r}k}/f_{\mathrm{e}k}$. The values $f_{\mathrm{r}k}$ are a splitting of f_{R} calculated for each k as the number of local iterations of R_{k+1} in the context of R_k. The values $f_{\mathrm{e}k}$ are a splitting of f_{E} calculated by $f_{\mathrm{e}k} = gcd(f_{\mathrm{r}k}, f_{\mathrm{E}k})$, where $f_{\mathrm{E}1} = f_{\mathrm{E}}$ and $f_{\mathrm{E}k} = f_{\mathrm{E}k-1}/f_{\mathrm{e}k-1}$, for $k = 2 \ldots N$.

Having thus inserted the upsampling chain to region R_N for each internal user X_j, we insert an extra upsampling actor S_j with upsampling rate s_j equal to local solution $\#_{\mathrm{L}} X_j$ in the context of region R_N (see Fig. 4).

Our PDN insertion algorithm [6] first inserts all the emitters, then visits the hierarchy nodes $\mathcal{P}_i$ and connects the users in $\mathcal{R}(\mathcal{P}_i)$ to the emitters as described above. The hierarchy tree is visited in a bottom-up order. Although U_k and S_j are new users, they can be located only in the current or higher nodes, so, in the end, all users are connected.

After inserting the PDN, the final step is to shortcut all the samplers with rate 1. The graph of Fig. 2 with its PDN is shown in Fig. 5. We obtain a bounded and safe SPDF graph where all communications are done via FIFOs. It can be dynamically scheduled in bounded memory [8].

B. Quasi-static scheduling

In SPDF, since the firing counts of some actors can be parametric, so is the schedule, which is said to be *quasi-static* [3]. Currently, our quasi-static scheduling algorithm requires that all parameters p_i can be ordered such that their corresponding modifiers $M(p_i)$ with periods $\alpha(p_i)$ are related by: $\frac{\#M(p_{i+1})}{\alpha(p_{i+1})} = f_i \cdot \frac{\#M(p_i)}{\alpha(p_i)}$, for some $f_i \in \mathcal{F}$. Observing that $\frac{\#M(p_i)}{\alpha(p_i)}$ denotes the modification count of p_i during a global iteration, we can expect our requirement to hold often in practice. A typical streaming application can be represented by nested loops where each parameter is modified exactly once at a certain loop level. The ordering of parameters corresponds to the different loop levels.

Our liveness criterion implies that we can ignore the edges with initial tokens and consider the corresponding acyclic graph. First, for the source (*i.e.,* non-PDN) part of the graph, we generate a string composed of the actors of that graph sorted topologically, e.g., ABC for Fig. 5.

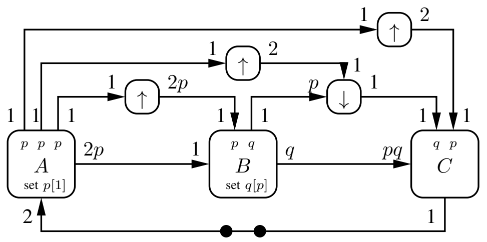

Fig. 5. The SPDF graph of Fig. 2 with its PDN.

In this string, we replace every actor X by the wrapper:

$$(\text{set } p_{i_1}; (\text{set } p_{i_2}; \ldots (\text{set } p_{i_N}; X^{f'_{N+1}})^{f'_N} \ldots)^{f'_2})^{f'_1}$$

where p_{i_k} ($k = 1 \ldots N$) are parameters used or modified by X; i_k are the increasing indexes of parameters in the above ordering; 'set p_i' sets a new parameter value for the given actor; $f'_1 = \#M(p_1)/\alpha(p_1)$; $f'_k = \frac{\#M(p_{i_k})/\alpha(p_{i_k})}{\#M(p_{i_{k-1}})/\alpha(p_{i_{k-1}})}$ for $k = 2 \ldots N$; $f'_{N+1} = \frac{\#X}{\#M(p_{i_N})/\alpha(p_{i_N})}$. For Fig. 5, we produce $(\text{set } p; A;)$ $(\text{set } p; (\text{set } q; B^p)^2)$ $(\text{set } p; (\text{set } q; C)^2)$.

Finally, we introduce modifier-to-user communication statements, equivalent to PDNs [6]. The modifier is implicitly connected to each user by a separate queue. It writes to all the queues with a single "push p_i". Each user reads the parameter values by a "pop p_i". In the wrapper for actor X, we replace the "set p_i" by "push p_i" if X is the modifier or by "pop p_i" otherwise. The push are moved *after* the actor invocation, because the actor as modifier has to compute the value to be pushed. In our running example, we get: $(A; \text{push } p)$ $(\text{pop } p; (B^p; \text{push } q)^2)$ $(\text{pop } p; (\text{pop } q; C)^2)$.

V. Case Study

We have applied SPDF to realistic case studies provided by an industrial partner. Figure 6 shows a SPDF model for a video decoder. The actor "**input**" reads the coded input frame and triggers a variable-length decoder "**vld**" for the 100 macroblocks of the frame. Once per frame (period 100) "**vld**" determines parameter p indicating whether the frame uses motion compensation. The actor "**mv**" determines whether the current macroblock has motion vectors (parameter t). If both conditions hold ($p \cdot t$), motion compensation is performed by the actor "**mc**". The actor "**vld**" triggers the calculation of four luminance blocks, "**lum**", each one computing an

l indicating whether it is coded. For coded blocks, inverse discrete cosine transform (IDCT) is performed by the actor "**l-idct**". The actor "**vld**" also determines whether chrominance is coded in macroblock (parameter c). If so, it triggers the execution of IDCT, "**c-idct**", followed by upscaling, "**upsc**", which builds four chrominance blocks out of one. Finally, the four luminance and chrominance blocks of the macroblock are converted one-by-one to RGB color format by the actor "**color**" and sent to the output frame. For each 100 macroblocks, the output frame expects 400 blocks.

Concerning rate consistency, the cycle condition is true for three undirected cycles, so the balance equation algorithm succeeds. Concerning safety, our hierarchy computation algorithm finds three disjoint nodes with parameter sets $\mathcal{P}_1 = \{p, t\}$, $\mathcal{P}_2 = \{l\}$, and $\mathcal{P}_3 = \{c\}$. The video decoder does not have directed cycles and the modifiers are located upstream to the users, so the liveness criterion holds.

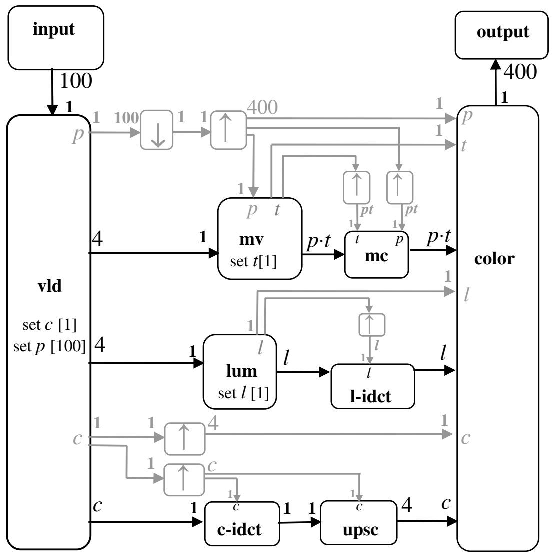

Fig. 6. Video decoder (compiler-inserted elements shown in gray).

Then, the PDN is inserted, shown in grey in Fig. 6. Finally, the quasi-static scheduler examines the periods of the modifiers and sorts the parameters: $(p \text{ (modified } \times 1/\text{frame)}, c\ (\times 100 \text{ more}), l, t\ (\times 4 \text{ more}))$. Applying our algorithm, we obtain the following schedule:

$$\begin{array}{l} \textbf{input}\quad ((\textbf{vld}; \text{push } c)^{100}; \text{push } p) \quad (\text{pop } p; (\textbf{mv}; \text{push } t)^{400}) \\ (\text{pop } p; (\text{pop } t; (\textbf{mc})^{pt})^{400}) \quad ((\textbf{lum}; \text{push } l)^{400}) \\ ((\text{pop } l; (\textbf{l-idct})^{l})^{400}) \quad (\text{pop } c; (\textbf{c-idct})^{c})^{100} \\ (\text{pop } c; (\textbf{upsc})^{c})^{100} \quad (\text{pop } c; (\text{pop } l; \text{pop } t; \textbf{color})^{4})^{100} \ \textbf{output} \end{array}$$

Actually, all the parameters (p, t, l, c) have been encoded as booleans. For space reasons, we have not presented this extension, but the whole methodology presented in this paper applies to this example without restrictions [6].

VI. Conclusions

We presented SPDF, a novel MoC for parametric streaming applications enabling static analysis and scheduling. We formulated sufficient and general static criteria for boundedness and liveness. In SPDF, parameter changes are allowed even within iterations. Their safety can be checked and their implementation is made explicit. All this was possible because we could manipulate and compare dynamic values by well-defined static operations on symbolic expressions. The same holds for quasi-static scheduling, which is the first step towards code generation for multi-core systems.

The most closely related MoC is PSDF [3], which requires to manually find the hierarchy levels and enclose them into hierarchical actors, *e.g.,* four levels for Fig. 6. With PSDF, the analysis is not completely static, as [3] applies a run-time analysis at hierarchy boundaries. The hierarchy analysis proposed in [9] requires significant manual help. The Scenario-Aware Data-Flow (SADF) MoC [10] extends SDF with performance analysis; yet, SADF does not define any boundedness analysis if hierarchy is present. The Variable-Rate Data-Flow (VRDF) MoC [11] introduced support for frequent changes of actor rates. However, VRDF imposes strong structural constraints on the graph. In particular, each production of p tokens must be matched by exactly one consumption of p tokens, and these pairs must be well parenthesized in the VRDF graph.

Multiprocessor scheduling for SPDF is an obvious and important extension of our work. Other important future work is SPDF scheduling with dynamic voltage and frequency scaling and performance-memory trade-off exploration. We also intend to explore other forms of dynamicity, such as dynamic graph reconfigurations, while preserving static schedulabilty.

Acknowledgments

Thanks are due to Ali-Erdem Ozcan for providing the case study and numerous discussions.

References

[1] S. S. Bhattacharyya, G. Brebner, J. W. Janneck, J. Eker, C. von Platen, M. Mattavelli, and M. Raulet, "OpenDF: a dataflow toolset for reconfigurable hardware and multicore systems," *SIGARCH Comput. Archit. News*, vol. 36, pp. 29–35, Jun. 2009.

[2] G. Kahn, "The semantics of a simple language for parallel programming," in *Information processing*. North Holland, 1974, pp. 471–475.

[3] B. Bhattacharya and S. S. Bhattacharyya, "Parameterized dataflow modeling of DSP systems," in *ICASSP'00*. IEEE, 2000.

[4] E. Lee and D. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. 36, no. 1, pp. 24–35, Jan. 1987.

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

[6] P. Fradet, A. Girault, and P. Poplavko, "Spdf: A schedulable parametric dataflow graph model (ext. version)," INRIA, Tech. Rep. 7828, 2011.

[7] J. Buck and E. Lee, "Scheduling dynamic data-flow graphs with bounded memory using the token flow model," in *ICASSP'93*, vol. I. Minneapolis (MN), USA: IEEE, Apr. 1993, pp. 429–432.

[8] T. M. Parks, "Bounded scheduling of process networks," Ph.D. dissertation, University of California, Berkeley, 1995.

[9] S. Neuendorffer and E. Lee, "Hierarchical reconfiguration of dataflow models," in *MEMOCODE'04*. IEEE, 2004, pp. 179–188.

[10] S. Stuijk, M. Geilen, B. Theelen, and T. Basten, "Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications," in *IC-SAMOS'11*. IEEE, 2011, pp. 404–411.

[11] M. Wiggers, M. Bekooij, and G. Smit, "Buffer capacity computation for throughput constrained streaming applications with data-dependent inter-task communication," in *RTAS'08*. IEEE, 2008, pp. 183–194.