BPDF: Boolean Parametric Data Flow

Vagelis Bebelis, Pascal Fradet, Alain Girault, Bruno Lavigueur

BPDF: Boolean Parametric Data Flow

Vagelis Bebelis , Pascal Fradet , Alain Girault , Bruno Lavigueur

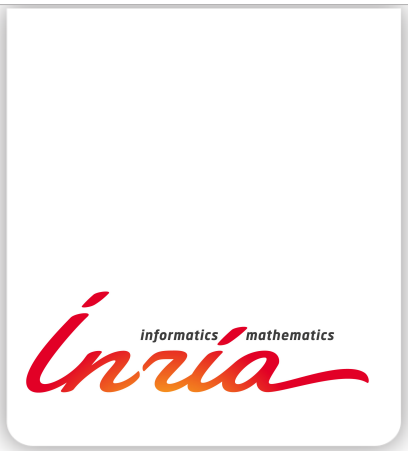
RESEARCH REPORT

N° 8333

July 2013

Project-Teams SPADES

ISSN 0249-6399 ISRN INRIA/RR--8333--FR+ENG



BPDF: Boolean Parametric Data Flow

Vagelis Bebelis * †, Pascal Fradet *, Alain Girault *, Bruno Lavigueur †

Project-Teams SPADES

Research Report n° 8333 — July 2013 — 21 pages

Abstract: Dataflow programming models are well-suited to program many-core streaming applications. However, many streaming applications have a dynamic behavior. To capture this behavior, parametric dataflow models have been introduced over the years. Still, such models do not allow the topology of the dataflow graph to change at runtime, a feature that is also required to program modern streaming applications. To overcome these restrictions, we propose a new model of computation, the *Boolean Parametric Data Flow* (BPDF) model which combines integer parameters (to express dynamic rates) and boolean parameters (to express the activation and deactivation of communication channels). High dynamicity is provided by integer parameters which can change at each basic iteration and boolean parameters which can even change within the iteration.
The major challenge with such dynamic models is to guarantee liveness and boundedness. We present static analyses which ensure statically the liveness and the boundedness of BDPF graphs. We also introduce a scheduling methodology to implement BPDF graphs on highly parallel platforms. Finally, we demonstrate our approach using a video decoder case study.

Key-words: Dataflow model of computation, Dynamicity, Parameters, Liveness, Boundedness, Scheduling

* INRIA
† STMicroelectronics

RESEARCH CENTRE GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

BPDF: Boolean Parametric Data Flow

Résumé : Les modèles de calcul flots de données sont bien adaptés à la programmation des applications de streaming sur les architectures multi-cœurs. Or, de nombreuses applications de streaming ont un comportement dynamique. Afin de prendre en compte cette dynamicité, plusieurs modèles de calcul paramétriques ont été proposés au cours des années récentes. Toutefois, ces modèles ne permettent pas de prendre en compte les reconfigurations dynamiques de la topologie d'un réseau flots de données, ce qui est requis dans les applications de streaming. Afin de résoudre ce problème, nous proposons un nouveau modèle de calcul, le modèle BPDF (« Boolean Parametric Data Flow »), qui combine des paramètres entiers (pour représenter les taux d'entrées sorties dynamiques) et des paramètres booléens (pour représenter l'activation et la désactivation des canaux de communication). La dynamicité est vient du fait que les paramètres entiers peuvent changer à chaque itération, et du fait que les paramètres booléens peuvent même changer au sein d'une itération.

Le principal défi avec de tels modèles dynamiques de calcul est de garantir les propriétés de vivacité et de bornage. Nous présentons des analyses statiques qui permettent de garantir statiquement la vivacité et le bornage d'un réseau BPDF. Nous présentons également une méthode d'ordonnancement afin de mettre en œuvre des réseaux BPDF sur des plateformes hautement parallèles. Enfin, nous illustrons notre approche avec une étude de cas d'un décodeur vidéo.

Mots-clés : Modèle de calcul flot de données dynamicité, paramètres, vivacité, bornage, ordonnancement

1 Introduction

Dataflow models of computation, such as SDF [10], have been very popular for designing streaming applications. These models allow static analyses to guarantee the boundedness and liveness of an application. However, they generally lack the expressivity and dynamicity needed by modern streaming applications such as high definition video codecs. To overcome this limitation, several *parametric* dataflow models have been proposed over the years, for instance PSDF [3], SADF [13], VRDF [16], or SPDF [7]. In contrast to SDF, these models allow the production and consumption rates of dataflow actors to change at runtime according to the values of the manipulated data (for instance the inputs).

Still, this is not enough because these parametric models do not allow the *topology* of the dataflow graph to change at runtime, a feature that is also required to program modern streaming applications. We propose a new parametric model of computation (MoC), called Boolean Parametric Data Flow (BPDF), allowing dynamic changes of the graph topology. In short, as in other parametric dataflow models, each BPDF actor has input ports (resp. output) labeled with a production rate (resp. consumption) that can be parametric (a product of integers and symbolic parameters). Integer parameters are allowed to change at runtime, between two iterations of the BPDF graph. Moreover, each BPDF edge can be annotated with a boolean expression defined using boolean parameters that are allowed to change at runtime, even *inside* an iteration of the graph. When a boolean expression is false, the edge it annotates is considered disabled (absent). Therefore, the topology of the BPDF graph changes according to the values taken by the boolean parameters.

As with the previous work on parametric models, the major challenge with BPDF is to allow more dynamicity while preserving liveness and boundedness guarantees. We propose syntactic criteria and algorithms to solve symbolically the balance equations, to ensure boundedness and to check that the number of initial tokens is sufficient to guarantee the liveness of a BPDF graph with cycles.

Because of the expected quality of service (e.g., high definition video), modern streaming applications are executed on many-core chips. Yet, the parallel implementation of parametric dataflow applications on such platforms remains a major challenge. We provide a scheduling algorithm for BPDF graphs targeted towards the new Sthorm many-core chip of STMicroelectronics [2]. In summary, our contributions consist in

- a novel model of computation called Boolean Parametric Data Flow (BPDF);
- algorithms to guarantee the boundedness and liveness of BPDF graphs;
- an approach to schedule BPDF graphs on manycore execution platforms;
- a case study based on the recent video codec VC-1.

The paper is organized as follows. Section 2 introduces the terminology of dataflow using SDF and presents our BPDF model. In Section 3, we describe the syntactic criteria and static analyses used to ensure boundedness and liveness. Section 4 presents the implementation of parameter communication and a method to schedule BPDF applications on many-core platforms. The model and its parallel implementation are illustrated in Section 5 by a case study. Finally, we summarize our contributions in Section 6.

2 Model of computation

We first present SDF (synchronous dataflow [10]), one of the simplest dataflow MoC. Then, we introduce our model – BPDF (Boolean Parametric DataFlow) – as an extension of SDF with integer and boolean parameters.

2.1 Synchronous Data Flow Model

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 all its incoming edges (its *inputs*) and produces data tokens to all 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. An actor can fire only when *all* its input edges have enough tokens (*i.e.,* at least the number specified by the rate). In SDF, all rates are constant integers, therefore known at compile time.

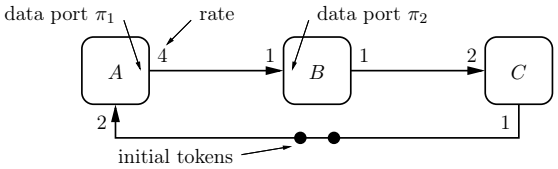


Figure 1: A simple SDF graph.

Figure 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*. For instance, edge (C, A) in Figure 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 initial state after a certain sequence of firings (ensuring boundedness of the FIFOs). Such sequence is called an *iteration* and can be repeated an infinite number of times (in the case of a reactive system).

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, in an iteration, 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 Figure 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 Figure 1 is rate-consistent, and the solutions are: $\#A = 1$, $\#B = 4$ and $\#C = 2$. Such a set of solutions forms a *repetition vector* noted $[\#A, \#B, \#C] = [1, 4, 2]$. 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 Figure 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 required 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 $A(B^2C)^2$.

2.2 Boolean Parametric Data Flow Model

We extend SDF by allowing rates to be parametric and edges to be annotated with a boolean condition. BPDF rates are products of positive integers (k) or symbolic variables (p). They are defined by the grammar:

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

with $\mathcal{P}_i$ denoting a set of symbolic variables that are the integer *parameters*. To ensure boundedness, a programmer using BPDF must specify a maximal value for each parameter. We do not fix specific values in the examples but we will denote the maximum of a parameter p as p_{max}; any given parameter p belongs to the interval $[1..p_{max}]$. We restrict integers to non-zero values to avoid division by zero when it comes to the analysis of the graph. This way, we strictly separate the role of rates (directly linked with the graph iteration) from the role of conditions (disabling edges and topology changes).

Each BPDF edge is annotated by a boolean condition which deactivate the edge when it evaluates to false. These boolean expressions are defined by the grammar:

$$\mathcal{B} \quad ::= \quad \texttt{tt} \mid \texttt{ff} \mid b \mid \neg\mathcal{B} \mid \mathcal{B}_1 \wedge \mathcal{B}_2 \mid \mathcal{B}_1 \vee \mathcal{B}_2$$

where b belongs to the set of symbolic variables $\mathcal{P}_b$ denoting boolean parameters.

Unlike the rates of SDF graphs that are fixed at compile time, the parametric rates of a BPDF graph can change dynamically between iterations. This change can be performed by a single actor or a centralized scheduler. This choice is implementation dependent and does not interact with the analyses or compilation process.

Moreover, each boolean parameter is modified by a single actor called its *modifier*. In BPDF, a modifier may change a boolean parameter within iterations using the annotation $b@\alpha$ where b is the boolean parameter to be set and α is the *period* of changes. The period of a boolean parameter b is the exact (possibly symbolic) number of firings of its modifier between two changes.

Intuitively, a BPDF actor reads and/or writes boolean parameters at specific periods. When it fires, it first evaluates the condition of its edges according to the current value of the boolean parameters. Then, it produces (resp. consumes) tokens on its outgoing (resp. incoming) edges that are annotated by a true condition only. It implies that a completely disconnected actor, *i.e.,* whose edges are all annotated by false (`ff`), fires (at least conceptually) but does not read nor write any channel except for reading or writing boolean parameters. It can do so until one of boolean parameter changes and sets one of the edge conditions to true (`tt`).

Boolean parameter communication must ensure that the parameters are written and read at the right pace without introducing deadlocks. The implementation of the boolean parameter communication is described in Section 3.3 and refined in Section 4.1.

Figure 2 shows a simple BPDF graph where actors have constant or parametric rates (e.g., p for the output rate of A). Omitted rates and conditions equal to 1 and `tt` respectively. Without going into details, the repetition vector is $[2, 2p, p, 2p, 2p]$ (see Section 3.1).

The edges (B, D), (B, C) and (C, E) are *conditional*. They are present only when their condition (here b or $\neg b$) is true. A sample execution of the graph is the following: A fires and produces p tokens on edge (A, B). Then B fires and sets the value of boolean parameter b. If b is true, B does not produce tokens on edge (B, D). As the edge is disabled, D fires twice without consuming tokens. B will fire a second time without changing the value of b enabling C to fire once. Finally, E will consume the tokens produced by C and D. If b is set to false, C is disconnected and it will fire without producing or consuming tokens. D and E will fire as expected. This continues until each actor has fired a number of times equal to its repetition count (as in SDF).

Further details on the execution of the model as well as its constraints that ensure boundedness and liveness are presented in Section 3.

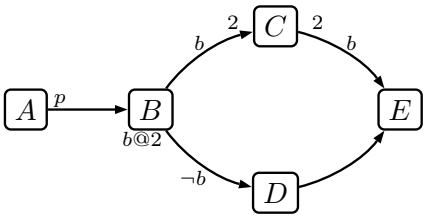


Figure 2: A simple BPDF graph with integer parameter p and boolean parameter b

Formally, a BPDF graph is defined as a 9-tuple $(\mathcal{G}, \mathcal{P}_i, \mathcal{P}_b, \beta, \iota, \delta, \rho, M, \alpha)$ 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;
- $\mathcal{P}_i$ and $\mathcal{P}_b$ are sets of integer and boolean parameters;
- $\beta : \mathcal{E} \to \mathcal{B}$ and $\iota : \mathcal{E} \to \mathbb{N}$ associate each edge with its condition and number of initial tokens, respectively;
- $\delta : \mathcal{P}_i \to \mathbb{N}^*$ returns the maximal values of each integer parameter;
- $\rho : \mathcal{A} \times \mathcal{E} \to \mathcal{R}$ returns for each port (represented by an actor and one of its edges) its associated (possibly symbolic) rate;
- $M : \mathcal{P}_b \to \mathcal{A}$ and $\alpha : \mathcal{P}_b \to \mathcal{R}$ return for each boolean parameter its modifier and its change period, respectively.

A user of a boolean parameter b is an actor connected by an edge whose condition involves b.

Definition 1. *The set of users of b, written $Users(b)$, is defined as $Users(b) = \{A, B \mid (A, B) \in \mathcal{E}, b \in \beta(A, B)\}$*

In addition of periods, we also use the dual notion of *frequency* which is the number of times a parameter may change within an iteration.

Definition 2. *The frequency of a parameter b , written $freq(b)$, is defined as $freq(b) = \frac{\#M(b)}{\alpha(b)}$.*

BPDF combines parametric rates and frequent topology reconfiguration as no other dataflow model proposes. Furthermore, as shown in the next section, this gain in expressivity does not prohibit effective static analyses.

3 Static Analyses

This section presents the three static analyses needed to ensure boundedness and liveness of BPDF graphs. In Section 3.1, we check rate consistency by adapting the analysis of SDF to BPDF. Conditions for consistency and solutions of balance equations are computed in terms of symbolic expressions. In Section 3.2, we check that the change periods of each boolean parameter are safe and show that, along with rate consistency, it ensures boundedness. Section 3.3 completes the analysis chain by checking for liveness.

3.1 Rate Consistency

As in SDF, we check the rate consistency of a BPDF graph by generating the associated system of balance equations. This system must be shown to have a non null solution for all possible values of parameters.

The equations are generated by considering the symbolic rates and by ignoring the boolean conditions on the edges. This enforces the system to be rate consistent for *all* possible configurations of the graph. Indeed, if the system is rate consistent when all edges are present (enabled), then it is also consistent when one or several edges are removed (disabled). Indeed, when removing edges, the resulting system of balance equations will be a subset of the system of equations of the fully connected graph. Checking rate consistency of all edges maybe considered too strict because it does not take into account the fact that some edges cannot be active at the same time (*e.g.,* two edges annotated by b and $\neg b$). On the other hand, it simplifies the understanding and implementation since a graph has a unique (although parametric) iteration vector.

We generalize the algorithm for solving the balance equations presented in [1] from SDF to BPDF by doing the same operations with symbolic factors. The algorithm relies on multiplication, division and greatest common divisor (*gcd*) of rates. These operations are easily expressed on $\mathcal{R}$ 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 parametric factors common to all solutions.

If the undirected version of the BPDF graph is acyclic, a solution to the balance equations always exists and can be found as follows. We arbitrarily set one of the solutions to 1 then recursively solve each equation. Finally, we normalize the solutions to integers. Consider, for instance, the graph of Figure2. By setting $\#A = 1$, we consecutively get:

$$\#B = p,\ \#C = \frac{p}{2},\ \#D = p,\ \text{and}\ \#E = p$$

To normalize the fractional solutions we multiply with the least common multiple of the denominators, in this case 2. Finally, we get the repetition vector

$$[\#A, \#B, \#C, \#D, \#E] = [2, 2p, p, 2p, 2p]$$

that is, $A^2B^{2p}C^pD^{2p}E^{2p}$ for short.

When the BPDF 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 $X_1, X_2, \ldots, X_n, X_1$ should satisfy the following condition:

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

where p_i and q_j denote the production and consumption rates of edge (X_i, X_j). This condition enforces that the product of "output" rates of a cycle should be equal to the product of "input" rates of this cycle.

Property 1 (Consistency)**.** *An BPDF graph is rate consistent if all its undirected cycles satisfy the cycle condition.*

Proof. To prove Prop. 1, we consider a cycle

$$X_1 \overset{p_1 \qquad q_2}{\longrightarrow} X_2 \overset{p_2}{\longrightarrow} \ldots \overset{q_n}{\longrightarrow} X_n \overset{p_n \qquad q_1}{\longrightarrow} X_1$$

We show that the solutions found for the tree obtained by removing the edge $X_n \overset{p_n \qquad q_1}{\longrightarrow} X_1$ are also solutions for the balance equation of the suppressed edge. The solutions verify the following balance equations:

$$\#X_i \cdot p_i = \#X_{i+1} \cdot q_{i+1} \qquad \text{for } i = 1 \ldots n-1$$

then multiplying all the *lhs* and *rhs* of the $n-1$ equations, we get

$$\#X_1 \cdot \ldots \cdot \#X_{n-1} \cdot p_1 \ldots \cdot p_{n-1} = \#X_2 \cdot \ldots \cdot \#X_n \cdot q_2 \ldots \cdot q_n$$

Removing the common factors yields

$$\#X_1 \cdot p_1 \cdot p_2 \ldots \cdot p_{n-1} = \#X_n \cdot q_2 \cdot \ldots \cdot q_n$$

Multiplying both sides by q_1, we get

$$\#X_1 \cdot q_1 \cdot p_1 \cdot p_2 \ldots \cdot p_{n-1} = \#X_n \cdot q_1 \cdot q_2 \ldots \cdot q_n$$

Then, using the cycle condition gives

$$\#X_1 \cdot q_1 \cdot p_1 \cdot p_2 \ldots \cdot p_{n-1} = \#X_n \cdot p_1 \cdot p_2 \ldots \cdot p_n$$

And, by simplifying by $p_1 \cdot p_2 \ldots \cdot p_{n-1}$, we finally get

$$\#X_1 \cdot q_1 = \#X_n \cdot p_n$$

which is the balance equation corresponding to the suppressed edge. The cycle condition guarantees that the balance equation of any suppressed edge is satisfied. Hence, the generic solutions satisfy the balance equations for all edges, which guarantees rate consistency of the graph. □

Rate consistency analysis either returns for each actor its (symbolic) minimal solution, or returns an unsatisfied cycle condition that can be used by the programmer to fix his BPDF graph.

3.2 Boundedness

If the BPDF graph returns to its initial state after each iteration, then all integer parameters can be modified at these points and boundedness is guaranteed. Without boolean parameters on the edges, rate consistency is sufficient to ensure that the graph returns to its initial state after each iteration. However, in BPDF, boolean parameters may change at a faster period by using the "$b@\alpha$" annotation. Yet, not all periods are safe and their consistency must be checked.

The criterion ensuring that parameter modification periods are safe relies on the notions of *regions* and *local iterations*. Intuitively, the criterion states that a parameter can be modified once per local iteration of its region. For Figure 2, the region of b consists of actors B, C, D, and E, so b can be changed after each local iteration $(B^2CD^2E^2)$[1], *i.e.,* after every 2 firings of B.

Definition 3 (Region)**.** *The region of a boolean parameter b, noted $\mathfrak{R}(b)$, is defined as:*

$$\mathfrak{R}(b) = \{M(b)\} \cup Users(b)$$

The *region* of b is the set containing its modifier and all its users. For example, the region of b in Figure 2 is $\mathfrak{R}(b) = \{B, C, D, E\}$.

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 4 (Local solutions)**.** *The local solution of an actor X_i in a subset of actors $L = \{X_1, \ldots, X_n\}$, written $\#_L X_i$, is defined as:*

$$\#_L X_i = \frac{\#X_i}{gcd(\#X_1, \ldots, \#X_n)}$$

For example, the global solutions of the actors of $\mathfrak{R}(b)$ in Figure 2 are $\#B = 2p$, $\#C = p$, $\#D = 2p$ and $\#E = 2p$. The *gcd* is p and their local solutions in $\mathfrak{R}(b)$ are $\#_{\mathfrak{R}(b)}B = 2$, $\#_{\mathfrak{R}(b)}C = 1$, $\#_{\mathfrak{R}(b)}D = 2$ and $\#_{\mathfrak{R}(b)}E = 2$. After one local iteration $B^2CD^2E^2$, all the edges influenced by b return to their initial state. Therefore, b can be changed between such local iterations. That is, B may modify b after 2 firings (as specified by the "$b@2$" annotation) and may do so $freq(b) = p$ times during the iteration. The actors C, D and E must read the new value of b after 1, 2, and 2 firings, respectively (and do so p times during the iteration).

However, local iterations can only be defined when the number of firings of each actor in the region of a parameter b is a multiple of the frequency of b (see Def. 2). This property is called *period safety.*

Definition 5 (Period Safety)**.** *A BPDF graph is period safe if and only if, for each boolean parameter $b \in \mathcal{P}_b$ and each actor $X \in \mathfrak{R}(b)$,*

$$\exists k \in \mathbb{N},\ \#X = k \cdot freq(b)$$

The factor k is the reading (or writing) period of b for X.

Period safety ensures that, during a local iteration of a region of a given boolean parameter, the number of tokens produced on any edge of this region equals the number of tokens consumed from this edge. It is ensured by a simple syntactic check on BPDF graphs.

[1]It is called local iteration of the region of b because the iteration $A^2B^{2p}C^pD^{2p}E^{2p}$ can be factorized as $A^2(B^2CD^2E^2)^p$.

In Figure 2, $\mathfrak{R}(b) = \{B, C, D, E\}$, $freq(b) = p$, and the solutions are $AB^{2p}C^pD^{2p}E^{2p}$. Each solution of the actors of $\mathfrak{R}(b)$ is a multiple of the frequency p. The annotation $b@2$ of B is thus period safe. Parameter b can be changed after each sub-sequence of firings $(B^2CD^2E^2)$.

Consider now the same graph but with the annotation $b@1$ instead. This graph is not period safe since the frequency of b becomes $2p$ whereas the solution of C is p (not a multiple of the frequency). Assume that B sets b to `tt` during its first firing and to `ff` for the rest of the iteration. Actor B will produce one token to the edge (B, C) and $(2p-1)$ tokens to (B, D). Actor C requiring two tokens will not fire and will not produce any tokens on (C, E), E will be blocked waiting for reading a token on (C, E) before being able to read the next value of b. As a consequence, tokens will accumulate on the edge (D, E). The graph would not return to its initial state. Actually, boundedness and liveness would not be guaranteed.

Property 2 (Boundedness)**.** *A rate consistent, period safe and live BPDF graph returns to its initial state at the end of its iteration.*

Proof. To prove Prop. 2, we consider an arbitrary edge

$$X \xrightarrow{x \quad f(b_1,\cdots,b_n) \quad y} Y$$

and show that during an iteration X produce the same number of tokens that Y consumes. Any edge returns to its initial state after one iteration and therefore the graph.

The condition $f(b_1, \cdots, b_n)$ is a boolean condition depending on n boolean parameters $b_1, b_2, ..., b_n$. Due to period safety, for each $b_i \in \{b_1, \cdots, b_n\}$ we have:

$$\exists k_i, l_i \in \mathbb{N}, \quad \#X = k_i \cdot freq(b_i) \quad \text{and} \quad \#Y = l_i \cdot freq(b_i) \tag{2}$$

By rate consistency, we also have $\#X \cdot x = \#Y \cdot y$. Therefore, for each b_i

$$k_i \cdot x = l_i \cdot y \tag{3}$$

When actor X (resp. Y) is fired, it produces x (resp. consumes y) tokens if $f(b_1, \cdots, b_n)$ and 0 otherwise. We write that X produces $prod(b_1, \cdots, b_n)$ and Y consumes $cons(b_1, \cdots, b_n)$ with

$$\begin{array}{rcl} prod(b_1, \cdots, b_n) & = & \text{if } f(b_1, \cdots, b_n) \text{ then } x \text{ else } 0 \\ cons(b_1, \cdots, b_n) & = & \text{if } f(b_1, \cdots, b_n) \text{ then } y \text{ else } 0 \end{array}$$

During an iteration, there will be $freq(b_i)$ (potentially different) values of boolean b_i. We note these values as the vector $\vec{b_i}$. Let $\uparrow$ be the upsampling operator defined as

$$[x_1, \ldots, x_n] \uparrow k = [\underbrace{x_1, \cdots, x_1}_{k}, \cdots, \underbrace{x_n, \cdots, x_n}_{k}]$$

Then, $\vec{b_i} \uparrow k_i$ is a vector such that the boolean at rank i represents the value that is used by X at its ith firing. The total production of tokens on the edge is:

$$P = sum\ (map_n\ prod\ [\vec{b_1} \uparrow k_1, \cdots, \vec{b_n} \uparrow k_n])$$

with

$$\begin{array}{rcl} sum\ [a_1, \cdots, a_n] & = & a_1 + \cdots + a_n \\ map_n\ f\ [a_{1,1}, \cdots, a_{1,m}] \cdots [a_{n,1}, \cdots, a_{n,m}] & & \\ & = & [f(a_{1,1}, \cdots, a_{n,1}), \cdots, f(a_{1,m}, \cdots, a_{n,m})] \end{array}$$

Equivalently the total token consumption is:

$$C = sum\ (map_n\ cons\ [\vec{b_1} \uparrow l_1, \cdots, \vec{b_n} \uparrow l_n])$$

It is easy to show the following properties on vector functions

$$(sum\ v)\ x = sum\ (v \uparrow x) \tag{4}$$
$$(map_n\ f\ [v_1 \cdots v_n]) \uparrow x = map_n\ f\ [v_1 \uparrow x \cdots v_n \uparrow x] \tag{5}$$
$$(v \uparrow x) \uparrow y = v \uparrow (x \cdot y) \tag{6}$$

and that, by rate consistency

$$sum\ (map_n\ prod\ v)\ x = sum\ (map_n\ cons\ v)\ y \tag{7}$$

Then,

$$\begin{array}{rcl}
P & = & sum\ (map_n\ prod\ [\vec{b_1} \uparrow k_1, \cdots, \vec{b_n} \uparrow k_n]) \\
& = & \frac{1}{x}\ x\ (sum\ (map_n\ prod\ [\vec{b_1} \uparrow k_1, \cdots, \vec{b_n} \uparrow k_n])) \\
& = & \frac{1}{x}\ (sum\ (map_n\ prod\ [\vec{b_1} \uparrow k_1 x, \cdots, \vec{b_n} \uparrow k_n x])) \\
& & \text{by (4), (5) and (6)} \\
& = & \frac{1}{x}\ (sum\ (map_n\ prod\ [\vec{b_1} \uparrow l_1 y, \cdots, \vec{b_n} \uparrow l_n y])) \\
& & \text{by (3)} \\
& = & \frac{y}{x}\ (sum\ (map_n\ prod\ [\vec{b_1} \uparrow l_1, \cdots, \vec{b_n} \uparrow l_n])) \\
& & \text{by (6), (5) and (4)} \\
& = & \frac{y}{x}\ \frac{x}{y}\ (sum\ (map_n\ cons\ [\vec{b_1} \uparrow l_1, \cdots, \vec{b_n} \uparrow l_n])) \\
& & \text{by (7)} \\
& = & C
\end{array}$$

Therefore, for any edge and any successive boolean values, the number of produced tokens is equal the number of consumed tokens in an iteration.

□

Since, during an iteration, the numbers of tokens produced and consumed on each edge are the same, the graph returns to its initial state. Rate consistency and period safety were crucial to ensure this property. However, we assumed that actors could be fired in the right order to respect dataflow and parameter communication constraints. This holds only when the graph is live and the next section shows how it is checked.

3.3 Liveness

In SDF, checking liveness is performed by finding a schedule for a basic iteration. Since each actor must be fired a fixed number of times in each iteration, this can be done by an exhaustive search. The situation is more complex in BPDF for two reasons:

- First, boolean parameters have to be communicated within the iteration, from modifiers to users. This introduces new constraints among firings of modifiers and users which may introduce deadlocks.
- Second, actors may have to be fired a parametric number of times during an iteration. Finding a schedule may, in general, involve some inductive reasoning.

Boolean parameter communication is implemented by adding to the BPDF graph new edges. For each parameter boolean b, we add between its modifier $M(b)$ and each user U of b an edge $M(b) \xrightarrow{u \quad m} U$, with u and m being the local solutions of U and $M(b)$ in the region of b:

$$u = \#_{\mathfrak{R}(b)}U \quad \text{and} \quad m = \#_{\mathfrak{R}(b)}M$$

In other words, $M(b)$ and U occur as $(\ldots M(b)^m \ldots U^u \ldots)$ in the local iteration corresponding to the region of b. It is easy to see that the solutions of the balance equations of the original graph are also valid for these new edges. During a local iteration, $M(b)$ will produce $m \cdot u$ copies of the value of b, which will all be read by U during this local iteration. We present in Section 4.1 a refinement of this implementation, which sends only one copy for each value of b.

This implementation allows modifiers to change the value of a boolean parameter even when the previously sent value has not been read. In this context, the effect of boolean parameters might be better described as disabling ports instead of edges. Indeed, at a given instant the input and output ports of an edge may be in a different state. The BPDF MoC ensures that the same number of tokens will be produced and consumed on each edge during an iteration (see Property 2).

The newly added communication edges may introduce new cycles in the graph, and subsequently deadlocks. As a consequence, liveness analysis must be performed on the BPDF graph augmented with all its communication edges. The fully connected graph includes all possible constraints, therefore if it is live, all resulting subgraphs with less edges (and thus less constraints) will be live too.

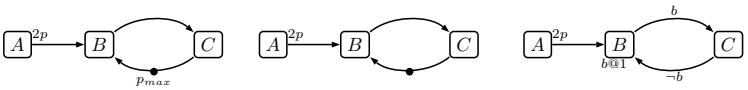
Acyclic BPDF graphs are always live. In such case, the topological order of the DAG defines a single appearance schedule [1]. For general graphs, a first simple criterion to ensure liveness is to check that every cycle contains a *saturated edge*. An edge (A, B) is said to be saturated if it contains enough initial tokens to fire B $\#B$ times. Since this edge has at least the total number of tokens consumed by B in a complete iteration, it does not introduce any constraints and can be ignored. If each cycle has a saturated edge, then the graph can be considered as acyclic and therefore live. The single appearance schedule corresponding to the topological order of the DAG obtained by removing all the saturated edges is also a schedule for the original graph.

When there are cycles without any saturated edge, we adapt the approach taken in SDF. Checking the liveness of cyclic SDF graphs is done by computing an iteration by abstract execution. Since the total number of firings is fixed, all possible ordering of firings can be tested. We adapt this approach to BPDF by

- ignoring booleans (all edges are assumed to be always enabled);
- testing all schedules where each consecutive firings of an actor A^s represents a non-parametric (integer) fractional of its number of firings in the iteration *i.e.,* $\exists k \in \mathbb{N}, \#A = k \cdot s$

Ignoring boolean conditions is safe since it maximizes constraints. If a schedule is found by assuming that all conditional edges are enabled, it will be also valid if some of these edges are disabled.

By considering only occurrences of the form A^s with $\#A = k \cdot s$, we bound the number of such occurrences. For instance, if $\#A = k \cdot p$, then at most k occurrences will be considered in the abstract execution. With this constraint, the liveness algorithm of SDF can be reused. We refer to this algorithm as Parametric SDF-like Liveness Checking (PSLC). For instance, the graph of Figure 3a (with repetition vector $AB^{2p}C^{2p}$) has a cycle without any saturated edge. Still, the PSLC algorithm finds the schedule $A(B^pC^p)^2$ so this graph is live.



(a) (b) (c)

Figure 3: Simple live BPDF graphs

Yet, there are cycles for which this approach is not sufficient. Consider, for example, the graph of Figure 3b. Its repetition vector is also $AB^{2p}C^{2p}$ and it is clearly live with the schedule $A(BC)^{2p}$. However, the PSLC algorithm cannot find it. In this case, a simple inductive reasoning would suffice. However, such an inductive approach gets complex to define in general. We continue by refining the SDF algorithm.

To deal with such problematic cycles, we use the standard clustering technique described in [1]. Clustering a subgraph $\mathcal{G}'$ of a graph $\mathcal{G}$ involves replacing $\mathcal{G}'$ by a single actor Z. The new actor Z is connected to the same external ports as $\mathcal{G}'$ was, but the port rates must be adjusted. The port rate r of an actor $A \in \mathcal{G}'$ is replaced by $r \cdot \#_{\mathcal{A}'}A$, where $\#_{\mathcal{A}'}A$ is the local solution of A in the set of actors $\mathcal{A}'$ of $\mathcal{G}'$.

In general, clustering arbitrary subgraphs can introduce cycles. Here, by clustering only cycles, we do not introduce new ones. For each cycle $C = X_1, \ldots, X_n$, our PSLC algorithm finds a local schedule. If this schedule does not fire the actors a non-parametric (integer) fractional of its total number of firings, the cycle is clustered into a new actor Z. The rate r of each port of an actor X_i connected to the rest of the graph is replaced by $r \cdot \#_C X_i$. It follows that a firing of Z corresponds to the firings $X_1^{k_1}, \ldots, X_n^{k_n}$ with $k_i = \#_C X_i$.

For instance, the local schedule for the cycle of the graph of Figure 3b is AB. Each actor is not fired a non-parametric (integer) fractional of its total number of firings ($2p$). The cycle is clustered into a new actor Z to get the new graph

$$A \xrightarrow{2p \quad 1} Z$$

The PSLC algorithm now finds the schedule AZ^{2p} which corresponds to $A(BC)^{2p}$ for the original graph, which is therefore correctly found to be live.

A final refinement is needed to take into account *false cycles*. For instance, the previous algorithm would fail to find a schedule for the cycle in Fig 3c since it does not have any initial token. However, it is clear that one of its two edges is always disabled; in other words, the cycle is false. We deal with this issue by using clustering. False cycles are detected using a truth table for all the conditions of the cycle. If, for each combination of values of the boolean parameters, at least one condition is false, then the cycle is false. Such false cycles can be clustered since there exists a schedule (possibly different) for each values of parameters (the graph is acyclic). Firing the resulting actor corresponds to executing a different schedule depending of boolean conditions. However, in all cases, the local schedule fires the actors the same number of times (*i.e.,* each X_i is fired $\#_C X_i$).

The false cycle of Fig 3c can be clustered into a new actor Z. A firing of Z corresponds to the schedule BC (*i.e.,* fire B then C) if $b = \texttt{tt}$, and to CB otherwise. The global schedule $(AZ)^{2p}$ is now easily found. It corresponds to the schedule $A(\text{if } b \text{ then } BC \text{ else } CB)^{2p}$ for the original graph.

To summarize, liveness checking proceeds by adding to the graph all the required boolean parameter communication edges, by suppressing the saturated edges, by detecting and clustering

all false cycles, by clustering all true cycles whose local schedule does not fire actors a fractional part of their global solutions, and finally by finding a schedule using the PSLC algorithm. If the last step succeeds, a sequential schedule has been found and the graph is live.

The above analysis is incomplete and there are live BPDF graphs that will be rejected, but it is sufficient in practice. Moreover, the above analysis does not try to estimate the minimal number of initial tokens for the graph to be live. The goal of the analysis is to verify that the graph is live, given an amount of initial tokens.

4 Implementation

The implementation of a bounded and live BPDF graph consists of two main tasks: parameter communication and scheduling.

4.1 Parameter Communication

Since integer parameters can only be changed between iterations, their communication is naturally synchronized and can be centralized. Hence, we focus here on boolean parameter communication, which requires synchronization between the firings of modifiers and users.

In Section 3.3, we implemented this synchronization by adding, for each parameter b with its modifier $M(b)$ and each user U of b, an edge $M(b) \xrightarrow{u \quad m} U$ with u and m being the periods of reading (by U) and writing (by $M(b)$) of b. With this pure dataflow implementation, if $u < m$, then each user U must wait for several firings of $M(b)$ before it can read the parameter and fire itself. This is more constrained than needed. The user could read the new value of a parameter just after one firing of its modifier.

We therefore propose a less constrained and more efficient implementation. Consider the following sub-graph:

$$R \xrightarrow{x \quad f(b_2) \quad r} \underset{b_1@a}{X} \xrightarrow{w \quad g(b_1,b_2) \quad y} M$$

Actor X is the modifier of b_1 and a user of b_2; it reads and writes tokens to and from conditional edges. The wrapper in Figure 4 implements one firing of X.

Actor X being the modifier of b_1, it must send a new value to each user of b_1 every a firings. Counter `b1` and instruction `write_p` are used for that purpose. The edges between X and users U_i are denoted by `U_i`. Actor X must also read b_2 every $\#X/\mathit{freq}(b)$ firings (the period safety ensures that this number is an integer). Counter `b2` implements this periodic reading. When it fires, X consumes r tokens on edge (R, X) if $f(b_2) = \texttt{tt}$ and produces w tokens on edge (X, W) if $g(b1, b_2) = \texttt{tt}$.

Such wrappers can be easily extended to implement writers and readers of any number of boolean parameters. Their role is to synchronize the firings of modifiers and users for parameter communication. A user will be blocked waiting for the modifier to produce the new boolean value that it needs. Thus, if a completely disconnected actor can fire without constraint, it will eventually have to wait in order to read (or write) a new boolean value.

This implementation can be described in a dataflow-like manner by adding edges between modifiers and users of the form

$$M \xrightarrow{\frac{1}{\alpha(b)} \qquad \frac{1}{u(b)}} U$$

```
// The counters for parameters b1 and b2
// are initialized to zero

// Write b1 at the right period
// for each user U_i
   if b1=0 then
           compute_new_b;
           write_p(b,U_1);...;write_p(b,U_n);
// Read b2 at the right period
   if b2=0 then read_p(b2,M);
  // Read r tokens on (R,X) if enabled
    if f(b2) then read(r,R);
    ...
  // Write w tokens to (X,W) if enabled
    if g(b1,b2) then write(w,W);
       ...
  // Increment the counters modulo
  // the writing and reading periods
      b1=(b1+1) mod a;
      b2=(b2+1) mod (#X/freq(b));
```

Figure 4: Wrapper implementation of a modifier and user

where $u(b)$ denotes the reading period of b by U and a fractional rate $\frac{a}{b}$ is means "produce/consume a tokens each b firings". A similar extension for SDF has been proposed in the fractional rate dataflow model [11].

4.2 Scheduling

In Section 3.3, we described a way to find sequential schedules for BPDF applications. However, our objective is to use BPDF to implement streaming applications on multi-cores with highly parallel schedules.

We focus on many-core platforms such as Sthorm developed by STMicroelectronics [2]. The native dataflow programming model of Sthorm uses the notion of *slots* to schedule the firing of actors. At the beginning of a slot, the controller selects several actors to be fired, and their execution takes place concurrently. When all previous executions are completed, the next slot starts. When the actors execute, the controller executes concurrently to select the next set of actors.

We describe here how to generate slotted, parallel, and as soon as possible (ASAP) schedules. We choose ASAP scheduling because it is highly parallel and typically used in streaming applications.

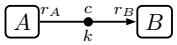
We assume that each actor is mapped onto a separate processing element of the many-core platform [2], and can be executed in parallel with other actors. An ASAP schedule is described by a set of constraints on the ordering of actor firings. A constraint is a relationship between two firing slots, of the form:

$$A_i > B_{f(i)} \tag{8}$$

where A_i denotes the slot of the ith firing of actor A, and $B_{f(i)}$ denotes the slot of the $f(i)$th firing of actor B (where f is any total function from $\mathbb{N}^*$ to $\mathbb{Z}$). The constraint simply states that A_i must be scheduled at a later slot than $B_{f(i)}$.

Consider the simple BPDF edge

[2]This is driven by the kind of applications we target, *e.g.*, high- definition codecs, which require very high performance that can only be achieved in hardware.



with production/consumption rates r_A and r_B, boolean guard c, and k initial tokens. The corresponding data dependency is captured by the following constraint:

$$B_i > A_{f(i)} \quad \text{if } c \qquad \text{with} \quad f(i) = \left\lceil \frac{r_B \cdot i - k}{r_A} \right\rceil \tag{9}$$

This constraint ensures that A has fired enough times so that, along with the k initial tokens, at least r_B tokens are present on the edge to allow B to fire. When c is false, the disabled edge does not enforce any constraint on B. These data dependencies (or dataflow constraints) are automatically generated for a BPDF graph.

Each edge of the BPDF graph produces one such dataflow constraint. Moreover, for each boolean parameter b, we add a constraint between its modifier $M(b)$ and each of its user U, of the form:

$$U_i > M(b)_{f(i)} \quad \text{with} \quad f(i) = (\left\lceil \frac{i}{\pi_r} \right\rceil - 1) \cdot \alpha(b) + 1 \tag{10}$$

where π_r denotes the reading period of the user ($\pi_r = \#U/freq(b)$). Such parameter communication constraint guarantees that the new values of the boolean parameter are produced by the modifier before being read by the users.

These constraints must be satisfied by any schedule. The parallel ASAP schedule is specified by taking, for each firing, the earliest slot satisfying all constraints. For each actor X, its constraints $X_i > A^1_{f_1(i)}, \ldots, X_i > A^n_{f_n(i)}$ are gathered in an equation of the form

$$X_i = \max(A^1_{f_1(i)}, \ldots, A^n_{f_n(i)}) + 1$$

which defines the ASAP slot that satisfies the constraints. In many cases, these equations can be simplified at compile time to produce quasi-static schedules. A detailed description of this process is out of scope of this paper. We present the main ideas on our running example.

Consider the graph in Figure 2, its dataflow constraints are:

$$B_i > A_{\lceil \frac{i}{p} \rceil}, \qquad C_i > B_{2i} \;\; if \;\; b, \qquad D_i > B_i \;\; if \;\; \neg b,$$

$$E_i > C_{\lceil \frac{i}{2} \rceil} if \;\; b, \qquad E_i > D_i$$

Its parameter communication constraints are:

$$C_i > B_{2i-1}, \qquad D_i > B_{2\lceil \frac{i}{2} \rceil - 1}, \qquad E_i > B_{2\lceil \frac{i}{2} \rceil - 1}$$

Finally, because each actor X is implemented as a hardware processing element, we have the implicit sequencing constraint $\forall i \in [1..\#X],\ X_i > X_{i-1}$, with the default value $X_0 = 0$. Gathering all these constraints for each actor, we get:

$$\begin{array}{lcll} A_i & = & A_{i-1} + 1 & i \in [1..2] \\ B_i & = & \max(A_{\lceil \frac{i}{p} \rceil}, B_{i-1}) + 1 & i \in [1..2p] \\ C_i & = & \begin{cases} \max(B_{2i}, B_{2i-1}, C_{i-1}) + 1 & if \;\; b \\ \max(B_{2i-1}, C_{i-1}) + 1 & if \;\; \neg b \end{cases} & i \in [1..p] \end{array}$$

And for $i \in [1..2p]$

$$\begin{array}{lcll} D_i & = & \begin{cases} \max(B_{2\lceil \frac{i}{2} \rceil - 1}, D_{i-1}) + 1 & \text{if } b \\ \max(B_i, B_{2\lceil \frac{i}{2} \rceil - 1}, D_{i-1}) + 1 & \text{if } \neg b \end{cases} & \\ E_i & = & \begin{cases} \max(B_{2\lceil \frac{i}{2} \rceil - 1}, C_{\lceil \frac{i}{2} \rceil}, D_i, E_{i-1}) + 1 & \text{if } b \\ \max(B_i, B_{2\lceil \frac{i}{2} \rceil - 1}, D_i, E_{i-1}) + 1 & \text{if } \neg b \end{cases} & \end{array}$$

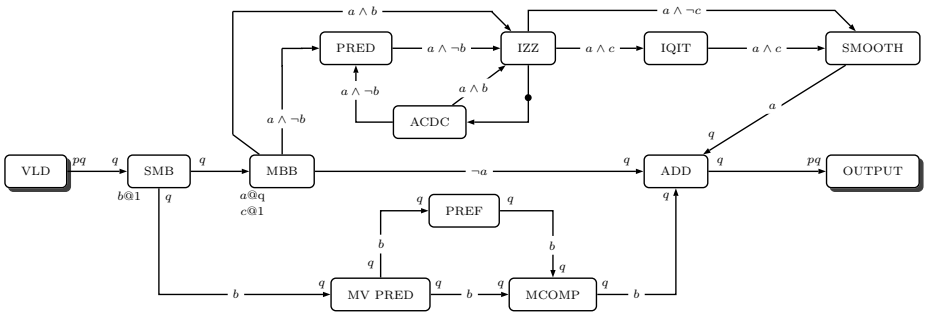


Figure 5: The VC-1 decoder captured in BPDF.

These functions are evaluated by the dynamic scheduler as soon as the parameters are known at runtime. They can also be simplified at compile time. Actor A fires only twice and its scheduling function can be inlined and simplified into $A_1 = 1$ and $A_2 = 2$. Reporting these values in the function B_i, we get $B_1 = 2$; for $i > 1$, the max function can be suppressed and we get $B_i = i+1$. Replacing B_i by $i + 1$ in function C_i yields:

$$C_i = \text{if } b \text{ then } 2i + 2 \text{ else } 2i + 1$$

Similarly for D, we end up with $D_i = i + 2$ in both cases b and $\neg b$. For E, the simplification cannot suppress all max functions. We get:

$$E_i = \text{if } b \text{ then } i + 4 \text{ else } \max(i + 2, E_{i-1}) + 1$$

Using these solutions, we can express the schedule of each actor as separate sequences of slots of the form:

$$\begin{aligned} A: &\quad \mathcal{F}\mathcal{F} \\ B: &\quad \mathcal{E}\mathcal{F}^{2p} \\ C: &\quad \mathcal{E}^2\big(b \ ? \ \mathcal{E}\mathcal{F} \ : \ \Phi\mathcal{E}\big)^p \\ D: &\quad \mathcal{E}^2\mathcal{F}^{2p} \end{aligned}$$

where $\mathcal{E}$ denotes no firing, Φ a dummy [3] firing of a disconnected actor, and $\mathcal{F}$ a real firing. The expression $(b \ ? \ E \ : \ F)$ executes the sequence E when b is true and F otherwise. For instance, C is not fired in the two first slots. Then, if b is true, it waits one slot and fires in the following slot; otherwise it performs a dummy firing and waits one more slot. This conditional sequence is repeated p times. The execution sequence of actor E cannot be put in this regular expression style and remains in a recursive form. These sequences can be easily combined and implemented as standard quasi-static schedules.

5 Case Study

In this section, the VC-1 video decoder is described in BPDF (Fig.5) and scheduled using the constraint framework. Each iteration of the graph corresponds to the computation of a single slice of a video frame.

[3] Dummy firing is a firing where the actor does nothing or is skipped altogether

The decoder is composed of two main pipelines, the inter and the intra. The inter pipeline is composed of actors MV PRED, PREF, and MCOMP, while the intra pipeline is composed of actors PRED, IZZ, ACDC, IQIT, and SMOOTH.

The decoder makes use of two integers and three boolean parameters. The integer parameters are p, which encodes the slice size in macroblocks, and q, which encodes the macroblock size in blocks. The boolean parameters capture whether a block is using intra (a) or inter (b) information and whether it is coded (c) or not. With these three boolean parameters, three possible modes of operation can be distinguished.

$$\begin{aligned} a \wedge \neg b : &\quad \text{Intra only} \\ \neg a \wedge b : &\quad \text{Inter only} \\ a \wedge b : &\quad \text{Intra and Inter} \end{aligned}$$

In the *Intra only* case, the value of the current block depends only on the values of the surrounding blocks. The inter pipeline is disabled. In the *Inter only* case, the value of the current block depends on the value of another block from a previous frame, as defined with a motion vector. Only the inter pipeline is used. Finally, in the *Intra and Inter* case, both pipelines are used but in the intra part the PRED actor is bypassed.

The case where both a and b are false is supposed not to occur: this configuration is invalid. The third boolean parameter c (coded) is used to bypass the inverse quantization and inverse transform (IQIT) when the block is not coded.

Two auxiliary actors are responsible for the boolean parameter modifications. Slice to MacroBlock (SMB) reads a slice, one macroblock at a time, and sets the macroblock parameter b. MacroBlock to Block (MBB) reads a macroblock, one block at a time, and sets the block parameters a and c.

Using the scheduling technique proposed in Section 4.2, the *as soon as possible* execution sequences of each actor are produced shown in Table 1. The complete schedule is the parallel combination of all sequences, which exhibits a high level of parallelism as there can be up to 8 actors executing concurrently.

An example of how the schedule is affected by the boolean parameters is given below. For the sake of simplicity we demonstrate only the change of boolean c in the intra pipeline:

```
...|b=tt|a=tt|    |    |       |       |...
...|    |c=tt|c=ff|    |       |       |...
...|SMB |MBB |MBB |... |...    |...    |...
...|... |    |IZZ |    |IZZ    |       |...
...|... |    |    |ACDC|       |ACDC   |...
...|... |    |    |IQIT|       |       |...
...|... |    |    |    |SMOOTH|SMOOTH|...
```

The VC-1 decoder could also be expressed *without* boolean parameters. However, in such a case, many actors would fire without performing useful computation. This would result in a much less efficient implementation of VC-1, since the extra firings would still consume both time and power to execute.

6 Conclusion

Several dataflow models supporting booleans have been proposed in the past, most notably the Boolean Data Flow (BDF) model [5]. The problem of bounded execution of a BDF graph is proved to be undecidable. Buck proposes a scheduling algorithm that uses a clustering technique

VLD	$\mathcal{F}$
SMB	$\mathcal{E}\mathcal{F}^p$
MBB	$\mathcal{E}^2\mathcal{F}^{pq}$
MV PRED	$\mathcal{E}^2(b\ ?\ \mathcal{F}\ :\ \Phi)^p$
PREF	$\mathcal{E}^3(b\ ?\ \mathcal{F}\ :\ \Phi)^p$
MCOMP	$\mathcal{E}^4(b\ ?\ \mathcal{F}\ :\ \Phi)^p$
PRED	$\mathcal{E}^3\big[((\neg b \wedge a)\ ?\ \mathcal{F}\ :\ \Phi)\mathcal{E}^2\big]^{pq}$
IZZ	$\mathcal{E}^2\big[((\neg b \wedge a)\ ?\ \mathcal{E}^2\ :\ \mathcal{E})\mathcal{F}\big]^{pq}$
ACDC	$\mathcal{F}\mathcal{E}\big((\neg b \wedge a)\ ?\ \mathcal{E}^2\ :\ \mathcal{E}\big)\big[\mathcal{F}\big((\neg b \wedge a)\ ?\ \mathcal{E}^2\ :\ \mathcal{E}\big)\big]^{pq-1}$
IQIT	$\mathcal{E}^2\big[\Big(a\ ?\ \big((\neg b\ ?\ \mathcal{E}^2\ :\ \mathcal{E})(c\ ?\ \mathcal{F}\ :\ \Phi)\big)\ :\ \Phi\Big)\big]^{pq}$
SMOOTH	$\mathcal{E}^2\big[\Big(a\ ?\ \big((\neg b\ ?\ \mathcal{E}^2\ :\ \mathcal{E})(c\ ?\ \mathcal{E}\mathcal{F}\ :\ \mathcal{F})\big)\ :\ \Phi\Big)\big]^{pq}$
ADD	$\mathcal{E}^2\big[(a\ ?\ (b\ ?\ \mathcal{E}\ :\ \mathcal{E}^2)(c\ ?\ \mathcal{E}^2\ :\ \mathcal{E})\ :\ \mathcal{E})^q\mathcal{F}\big]^p$
OUTPUT	$\mathcal{E}^2\big[(a\ ?\ (b\ ?\ \mathcal{E}\ :\ \mathcal{E}^2)(c\ ?\ \mathcal{E}^2\ :\ \mathcal{E})\ :\ \mathcal{E})^q\mathcal{E}\big]^p\mathcal{F}$

Table 1: ASAP execution sequences for each VC-1 actor

to produce bounded schedules for a subset of BDF graphs. Yet, the analysis is restricted to BDF graphs that admit a single appearance schedule and are free of directed cycles.

Another difference between BDF and BPDF is that, in BPDF, the graph iteration is cleanly defined using either constant values or integer parameters. Thus, the number of firings of each actor is known at the beginning of each graph iteration. The balance equations in BDF are solved using a parameter based on the boolean values, which leads to a boolean dependent graph iteration. This means that the number of firings of each actor is unknown until the end of the iteration.

Although, in some cases, a repetition vector can be produced for a BDF graph to guarantee a bounded memory execution, the clustering algorithm produces a schedule with no finite execution, disregarding the repetition vector and the graph iteration. In BPDF, the graph iteration can always be respected while the boundedness is guaranteed.

Integer Data Flow (IDF) [6] is an extension of BDF that uses integer parameters to introduce actors that act as `switch case` structures. This can be achieved with BPDF by using multiple boolean parameters. IDF suffers from the the same decidability limitations as BDF. This is because both models are Turing complete, in contrast with BPDF which is not as expressive.

Other related models include Cyclo-Static [4] and Cyclo-Dynamic Data Flow [15], $*$-Charts [8], Bounded Dynamic Data Flow with control flow [12], and dynamic configuration of PSDF graphs [9]. None of them provide any of the static guarantees that BPDF model does (mostly because the analyses depend on the boolean values). Some of these MoCs also lack a complete and formal presentation. Models such as PSDF [3], SADF [13], VRDF [16], or SPDF [7] propose parametric rates without dynamic topology changes. Our integer parameters, changing only between iterations, can be compared to those of PSDF. Our boolean parameters are closer to the parametric rates of SPDF which are also allowed to change within iterations. In contrast with BPDF, teleport messaging [14] uses parameters to change the internal functionality of the actors. BPDF uses parameters to change port rates and the graph topology.

We presented BPDF, a novel parametric dataflow model of computation for streaming applications with dynamic changes of rates and of the graph topology. We described static analyses to guarantee the bounded and live execution of BPDF graphs. Compared to existing models, BPDF provides increased expressiveness by combining integer parameters for the input and output rates of the actors, and boolean parameters for the enabling conditions of the edges. Integer

parameters can change between iterations, while boolean parameters, which allow the graph topology to change dynamically, can change also *during* an iteration.

Finally, we described a method to implement BPDF graphs onto many-core platforms while effectively utilizing the high parallelism that they provide. We demonstrated such a case with a modern video decoder, captured in BPDF and scheduled using the aforementioned method.

As future work, we want to explore the scheduling possibilities of BPDF extended with user-defined scheduling constraints. With these additional constraints, we can tweak the schedule and optimize several criteria like power consumption. Finally, an implementation on STMicroelectronics' many-core platform STHORM is in progress. We want to evaluate various scheduling techniques for several applications (*e.g.,* VC-1, H.264 and HEVC) on the platform.

References

[1] S. S. Battacharyya, E. A. Lee, and P. K. Murthy. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[2] L. Benini, E. Flamand, D. Fuin, and D. Melpignano. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *DATE*, pages 983–987, 2012.

[3] B. Bhattacharya and S. S. Bhattacharyya. Parameterized dataflow modeling for dsp systems. *IEEE Trans. on Sig. Proc.*, 49(10):2408–2421, 2001.

[4] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static dataflow. In *IEEE Trans. on Sig. Proc.*, volume 44, pages 397 – 408, February 1996.

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

[6] J. T. Buck. Static scheduling and code generation from dynamic dataflow graphs with integer-valued control streams. In *ALISOMARSCC'94*, volume 1, pages 508–513 vol.1, 1994.

[7] P. Fradet, A. Girault, and P. Poplavkoy. SPDF: A schedulable parametric data-flow moc. In *DATE*, pages 769–774, 2012.

[8] A. Girault, B. Lee, and E. A. Lee. Hierarchical finite state machines with multiple concurrency models. In *IEEE Trans. on Computer-Aided Design*, volume 18, pages 742–760, June 1999.

[9] D.-I. Ko and S. S. Bhattacharyya. Dynamic configuration of dataflow graph topology for dsp system design. In *IEEE ICASSP*, March 2005.

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

[11] H. Oh and S. Ha. Fractional rate dataflow model and efficient code synthesis for multimedia applications. In P. Marwedel and S. Devadas, editors, *LCTES-SCOPES*, pages 12–17. ACM, 2002.

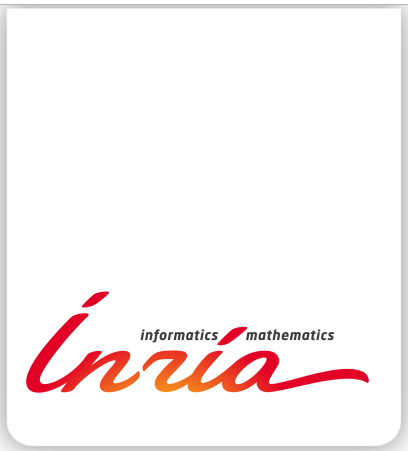
[12] M. Pankert, O. Mauss, S. Ritz, and H. Meyr. Dynamic data flow and control flow in high level dsp code synthesis. In *IEEE ICASSP*, volume ii, pages 449–452, 1994.

[13] B. Theelen, M. Geilen, T. Basten, J. Voeten, S. Gheorghita, and S. Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *MEMOCODE*, pages 185–194, Napa Valley (CA), USA, jul 2006. ACM-IEEE.

[14] W. Thies, M. Karczmarek, J. Sermulins, R. Rabbah, and S. Amarasinghe. Teleport messaging for distributed stream programs. In *PPoPP*, pages 224–235. ACM, 2005.

[15] P. Wauters, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-dynamic dataflow. In *Parallel and Distributed Processing*, pages 319–326, 1996.

[16] M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit. Buffer capacity computation for throughput constrained streaming applications with data-dependent inter-task commnication. *ACM Trans. Embedded Comput. Syst.*, 10(2):17, 2010.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399