A Framework to Schedule Parametric Dataflow Applications on Many-Core Platforms

Vagelis Bebelis ♠♣ Pascal Fradet ♠♡ Alain Girault ♠♡

INRIA ♠
Univ. Grenoble Alpes, F-38000, Grenoble, France ♡
STMicroelectronics ♣
first.last@inria.fr

Abstract

Dataflow models, such as SDF, have been effectively used to program streaming applications while ensuring their liveness and boundedness. Yet, industrials are struggling to design the next generation of high definition video applications using these models. Such applications demand new features such as parameters to express dynamic input/output rate and topology modifications. Their implementation on modern many-core platforms is a major challenge.

We tackle these problems by proposing a generic and flexible framework to schedule streaming applications designed in a parametric dataflow model of computation. We generate parallel as soon as possible (ASAP) schedules targeted to the new STHORM many-core platform of STMicroelectronics. Furthermore, these schedules can be customized using user-defined ordering and resource constraints.

The parametric dataflow graph is associated with generic or user-defined specific constraints aimed at minimizing timing, buffer sizes, power consumption, or other criteria. The scheduling algorithm executes with minimal overhead and can be adapted to different scheduling policies just by adding some constraints. The safety of both the dataflow graph and constraints can be checked statically and all schedules are guaranteed to be bounded and deadlock free. We illustrate the scheduling capabilities of our approach using a real world application: the VC-1 video decoder for high definition video streaming.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Data-flow languages; D.4.1 [*Process Management*]: Scheduling; D.2.4 [*Software/Program Verification*]: Formal methods

General Terms Algorithms, Languages, Verification

Keywords Dataflow, Manycore, Scheduling, Liveness, Boundedness

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
LCTES '14, June 12–13, 2014, Edinburgh, UK.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2877-7 /14/06…$15.00.
http://dx.doi.org/10.1145/2597809.2597819

1. Introduction

Dataflow models of computation, such as SDF [13], provide analyses to guarantee the boundedness and liveness of an application. However, they generally lack the expressivity needed by modern streaming applications such as next generation video codecs. *Parametric* dataflow models such as PSDF [6], VRDF [22], SADF [21], SPDF [8] or BPDF [3] allow more dynamicity while preserving liveness and boundedness guarantees.

The target for streaming applications is often modern embedded platforms which typically use many-core architectures with network-on-chip interconnection. Yet, the parallel implementation of parametric dataflow applications on such platforms remains a major challenge.

In this paper, we propose a framework for effectively producing parallel schedules for the next generation of streaming applications. We consider Boolean Parametric Data Flow (BPDF) [3] model of computation and the STHORM (formerly, P2012) [4] many-core platform by STMicroelectronics. BPDF is a very expressive parametric dataflow model that combines integer and boolean parameters allowing dynamic data rates and graph topology changes while providing static guarantees. STHORM is a leading-edge, cluster-based, many-core architecture, designed to support the future high definition video and augmented reality embedded applications.

We focus on the parallel scheduling of applications expressed as BPDF graphs. A BPDF graph consists of actors linked by dataflow edges (FIFOs), each actor producing a parametric number of tokens. Each edge may also have a boolean guard that enables or disables the edge at runtime. We consider *coarse-grain* BPDF applications, where actors are large blocks of C code, typically video codec filters. High-definition video codecs require very fast execution times, for this reason each actor is implemented as a hardware processing element or executes as software on a dedicated core.

We rely on a *slotted* scheduling model compatible with STHORM, such that, in each slot, several actors are scheduled to execute. Since each actor is a separate processing element, their execution can proceed concurrently. When all fired actors have terminated, new actors can then be scheduled in the next slot. This scheduling scheme is general enough and it can be used by other many-core platforms. This slotted scheduling contrasts with other existing multi-processor scheduling methods where the execution time of each actor plays a central role to determine the shortest schedule.

In applications such as video decoding, the complexity of filters depends on data and precise timing information cannot be known statically. In such a context, an ASAP execution of the available tasks is the best strategy[20]. When precise timing information

is available, slotted scheduling should strive to minimize slack between slots. Techniques such as retiming [15] can be used for that purpose.

Our scheduling procedure starts by deriving from the graph a set of graph constraints representing data dependencies. They express the partial ordering of the firings of the actors.

Additional ordering constraints can be added to tune the scheduling policy. For instance, constraints can be used to enforce properties inherent to the execution platform or optimize various criteria, such as buffer size or power consumption. Furthermore, it can be checked that these constraints preserve liveness.

Along with ordering constraints, our framework also supports resource constraints. These constraints filter the fireable actors at each slot to accommodate physical mapping on the platform or to take into account timing and power consumption. Resource constraints are expressed as a set of rules that regulate the parallel execution of actors.

In many cases, constraints can be statically simplified and scheduling entails only a minimal dynamic overhead. The scheduling algorithm finds, at each slot, the set of actors whose constraints are satisfied and can thus be fired. This amounts to an ASAP quasi-static slotted schedule. Static analyses guarantee that it exists and that it is bounded. The scheduler is executed in parallel with the previously issued actors so the overhead remains minimal.

Constraints make the approach *flexible* since the same scheduling algorithm can take into account a new platform or new optimization criteria just by modifying the set of constraints. Our approach focuses on flexibility, enabling easy manipulation and fine-tuning of the schedule for various platforms and optimizing criteria. In summary, our contributions consist in

- a flexible framework to schedule parametric dataflow applications on many-core platforms;
- several kinds of constraints (ordering and resource constraints) to specify optimized and tailor-made scheduling policies;
- a correct-by-construction approach that guarantees bounded and deadlock free schedules.

The paper is organized as follows. In Section 2, we introduce the technical context, namely BPDF and STHORM. In Section 3, we present the scheduling framework composed of different kinds of constraints and a simple ASAP scheduler. In Section 4, we illustrate our approach using the well-known VC-1 video codec [14]. Section 5 compares our approach to related work. Finally, Section 6 summarizes our contribution and hints at future research directions.

2. Background and Context

Our scheduling technique considers the recent Boolean Parametric Data Flow (BPDF) model [3] and a modern many-core platform (STHORM) [4].

2.1 Boolean Parametric Data Flow Model

BPDF can be described as a parametric extension of the SDF (Synchronous Data-Flow) model [13] that also allows dynamic changes of the topology. In SDF, an application is defined as a directed graph of *actors*. Each actor represents a functional unit and has *ports* connected by *edges* implemented as FIFO channels. Each time an actor executes (*fires*), it consumes data tokens from its incoming edges (its *inputs*) and produces data tokens on its outgoing edges (its *outputs*). The number of tokens produced and consumed are specified by *rates* associated with each port. In SDF, all rates are constant and known at compile time.

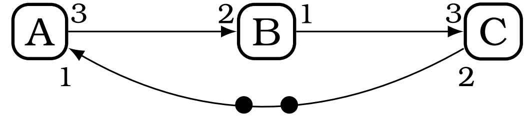

Figure 1. A consistent SDF graph

The *state* of an SDF graph is the number of tokens stored at each edge at a given instant. An edge can have zero or more tokens at any instant. The initial tokens at each edge specify the initial state of the graph.

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*, obtained by solving the so-called system of *balance equations*. This system is made of one equation per edge (X_1, X_2) of the form

$$\#X_1 \cdot r_1 = \#X_2 \cdot r_2 \quad (1)$$

where $\#X_1$ and $\#X_2$ indicate the number of firings of actors X_1 and X_2 for one iteration and r_1 and r_2 the rates of the equivalent ports.

A graph is *consistent* if its system of balance equations has non-trivial solutions. The repetition vector is the minimal solution of the balance equations. That vector represent the number of firings of each actor per iteration.

A simple SDF graph is shown in Fig. 1. The graph is consistent, has the repetition vector $[A^2\, B^3\, C^1]$ and the initial state $[0\,0\,2]$ for edges (A, B), (B, C) and (C, A) respectively. A sample schedule for an iteration is: $A\ B\ A\ B\ B\ C$.

BPDF extends SDF by allowing rates to be parametric and edges to be annotated with a boolean condition. BPDF port 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 the set of symbolic variables $\mathcal{P}_i$ denotes the integer *parameters*.

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:

$$\mathcal{B} \quad ::= \quad \mathit{true} \mid \mathit{false} \mid b \mid \neg\mathcal{B} \mid \mathcal{B}_1 \wedge \mathcal{B}_2 \mid \mathcal{B}_1 \vee \mathcal{B}_2$$

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. Moreover, each boolean parameter is modified by a single actor called its *modifier*. A modifier may change a boolean parameter within an iteration using the annotation $b@\alpha$ where b is the boolean parameter to be set and α is the *writing period*. The period of a boolean parameter b is the exact (possibly symbolic) number of firings of its modifier between two changes. Depending on the graph (rates, modifiers, users, ...) some writing periods are invalid. BPDF checks that periods are safe *i.e.,* that the graph returns to its initial state after each iteration.

The actors that have a boolean parameter on any of their edges are called *users* of that parameter. Users read new parameter values periodically. This period is measured in number of firings of the user and is called *reading period*. It is easily computed from the writing period and the number of firings of the user and modifier.

The number of different values produced for a parameter is called its *frequency*. BPDF guarantees that the users will use properly all the produced boolean values within an iteration.

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 $false$, fires (at least conceptually) but does not read nor write any channel except for reading or writing boolean parameters.

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 $true$ respectively. The symbolic solution of the balance equations gives a repetition vector which can be noted as $[A^2\ B^{2p}\ C^p\ D^{2p}\ E^{2p}]$. The actor B is the modifier of b with a writing period of 2. The actors (C, D, E) are users of b. Intuitively, the writing period of b is safe because the global iteration can be seen as A^2S^p where S is the sub-iteration $B^2CD^2E^2$. The modifier (resp. the users) writes (resp. read) b at each such sub-iteration *i.e.,* after each 2 firings for $B\ D\ E$ and after each firing for C. Note that a period of p would have been safe whereas 1 or 3 would have been invalid

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 iteration of the graph is the following. First, p is set and sent to users; A fires and produces p tokens on edge (A, B). Then B fires and starts by setting the value of the boolean parameter b.

- If b is *true*, B produces one token on (B, C) and does not produce tokens on (B, D). As the edge is disabled, D fires twice without consuming tokens and producing 2 on (D, E). Actor B fires a second time without changing the value of b enabling C to fire once and producing 2 token on (C, E). Finally, E can fire twice to consume the tokens produced by C and D.
- If b is *false*, C is disconnected and it will fire once without producing or consuming tokens. B will fire twice producing 2 tokens on (B, D) that will be consumed by two firings of D. The actor E will fire twice to consume the tokens produced by D since the edge (C, E) is disabled.

This sub-iteration continues until each actor has fired a number of times equal to its repetition count (as in SDF).

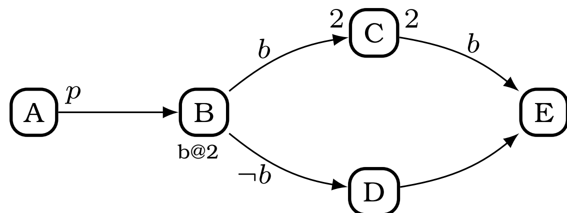

Figure 2. A simple BPDF graph with integer parameter p and boolean parameter b

BPDF combines parametric rates and frequent topology reconfigurations as no other dataflow model. This makes its scheduling on many-core platforms quite challenging. We show in the next sections that our scheduling framework is expressive and flexible enough to produce various parallel schedules for BPDF applications.

2.2 The STHORM Platform

The platform we target is STHORM by STMicroelectronics [4], which is representative of a modern many-core platform. It is composed of a set of clusters (currently up to 32) in a GALS design and connected with an asynchronous Network-on-Chip.

Each cluster contains up to 16 software processing elements (SWPE), as a general purpose RISC Processor, and a set of dedicated hardware processing elements (HWPE). In our implementation, each BPDF actor is implemented in a separate (hardware or software) processing element and the execution of the application (*i.e.,* the scheduling of the BPDF graph) is controlled by a processor.

Moreover, STHORM includes a native programming model that simplifies the parallel implementation of streaming applications. This programming model uses *filters* to implement applications. A filter can be:

- A *primitive filter* which applies a well defined function to a set of input data in order to produce a set of output data. It is the building block of STHORM's native programming model. We implement BPDF actors as primitive filters. A filter can execute on a HWPE or a SWPE.
- A *controller* which schedules the firing of the filters and controls configuration parameters for each filter.

We focus on the generation of the controller that controls the execution of each BPDF actor.

The native programming model uses the notion of *slots* to schedule the firing of the filters. At the beginning of a slot, the controller selects several filters to be fired, and their execution takes place concurrently. When all previous executions are completed, the next slot starts. The controller can execute concurrently with the filters and therefore the hardware pipeline is not slowed down by the controller.

We produce slotted schedules which can be directly implemented using this model. We believe that such a scheduling model can also be used by other state-of-the-art many-core platforms. For instance, modern Graphical Processing Units (GPUs) support a similar execution model. In mainstream GPU programming models such as CUDA [17] and OpenCL [16], the host processor, equivalent to the controller of STHORM, creates a task group, loads it on the GPU, and get the results when the latter has finished execution. In parallel with the execution of the task group, the host processor may determine tasks to be executed next.

Although, we rely on the STHORM platform, our scheduling framework can easily adapt to produce non slotted schedules as discussed in Sec. 3.4. Moreover, we assume that each actor is mapped on a separate processing element but the framework can handle any other kind of static assignment, where actors may share resources as shown in Sec. 3.2.

3. Scheduling Framework

Our scheduling framework takes as input an application (specified as a BPDF graph) and a set of user-defined constraints, and produces the ASAP slotted schedule meeting the constraints. An overview of the scheduling framework is presented in Fig. 3.

The constraints belong to two distinct types: *Ordering constraints* that restrict individual actor firings, and *resource constraints* that control parallel execution (*e.g.,* limiting the level of parallelism). The two types of constraints are presented in detail in Sec. 3.1 and 3.2.

Ordering constraints can derive either from the *application* expressing the data dependencies of the dataflow graph, or from the *user* expressing platform specificities or optimizing some crite-

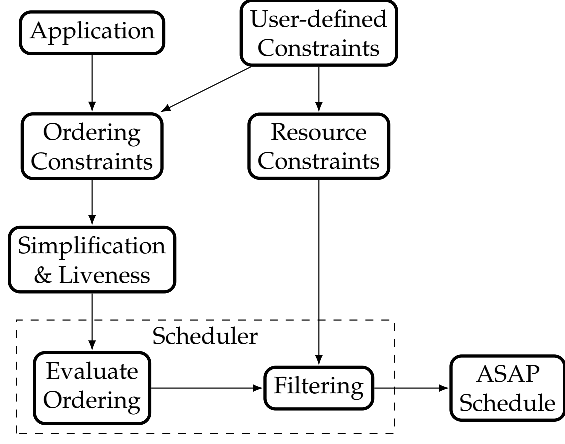

Figure 3. Scheduling Framework.

ria. Application constraints can only be ordering constraints. User-defined constraints can be both resource and ordering constraints.

Inconsiderate user-defined constraints may introduce deadlocks. To guarantee liveness, such constraints are automatically detected and rejected (see Sec. 3.3). The valid constraints can then be simplified and taken into account by the scheduler as described in Sec. 3.4.

3.1 Ordering Constraints

An ordering constraint is a relationship between the firings of two actors of the form:

$$A_i > B_{f(i)}$$

where A_i denotes the ith firing of actor A and $B_{f(i)}$ denotes the $f(i)$th firing of actor B (where f is any total function from $\mathbb{N}^*$ to $\mathbb{Z}$). A null or negative $f(i)$ means that instance A_i does not depend on $B_{f(i)}$. Some ordering constraints are derived from the application and additional ones can be given by the user.

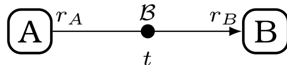

Figure 4. A generic BPDF edge.

3.1.1 Application constraints

Application constraints (or dataflow constraints) are automatically derived from data dependencies between actors. For each edge between actors A and B with production/consumption rates r_A and r_B respectively, initial tokens t, and boolean guard $\mathcal{B}$ (see Fig. 4), the following data constraint is generated

$$B_i > A_{f(i)} \quad \text{with} \quad f(i) = \left\lceil \frac{r_B \cdot i - t}{r_A} \right\rceil \tag{2}$$

If actor A has fired $\left\lceil \frac{r_B \cdot i - t}{r_A} \right\rceil$ times, it has produced $\left\lceil \frac{r_B \cdot i - t}{r_A} \right\rceil \cdot r_A$ tokens, which is greater than or equal to the number of tokens required to fire actor B i times, that is $r_B \cdot i - t$. Therefore, after $f(i)$ firings of A and $i-1$ firings of B, there remains enough tokens in the FIFO to fire B for the ith time.

Equation (2) does not depend on the boolean guard $\mathcal{B}$. However, the scheduler takes boolean guards into account by disregarding data constraints of disabled edges.

Boolean parameters introduce constraints due to the communication of their values between modifiers and users. A user needs to read a new value according to its reading period (π_r). The modifier produces a new value according to its writing period (π_w). Therefore, we get the following ordering constraint for each user (U) - modifier (M) pair:

$$U_i > M_{f(i)} \quad \text{with} \quad f(i) = \pi_w \cdot \left\lfloor \frac{i-1}{\pi_r} \right\rfloor + 1 \tag{3}$$

This states that the ith firing of U requires the boolean value that is produced on the $f(i)$th firing of M. U will use this value for its next π_r firings. The constraint restricts the user to wait for the production of a boolean value but does not restrict the modifier. Indeed, the modifier may produce a new boolean value (or all the boolean values) before the user has finished using the previous one. The user will use the new values later, based on its reading period, when they are needed.

3.1.2 User constraints

User constraints are typically used to optimize various criteria (*e.g.,* power consumption, buffer size) or express platform specificities (*e.g.,* resource limitations). They are defined by the programmer for a specific application or platform. Consider again the simple BPDF graph of Fig. 4.

The repetition vector is $[A^{r_B}\ B^{r_A}]$ and A will fire r_B times without constraints[1]. Since A fires r_B times consecutively, if B does not consume enough tokens, there will be an accumulation of tokens on the edge buffer.

If the programmer wants to restrict the buffer to be of size, say k, it can restrict the execution of A so that it fires only when there is enough space left on the buffer thanks to the constraint:

$$A_i > B_{g(i)} \quad \text{with} \quad g(i) = \left\lceil \frac{r_A \cdot i + t - k}{r_B} \right\rceil \tag{4}$$

If only application constraints are used, the dataflow analyses guarantee liveness and the existence of a valid schedule. When additional constraints are considered, they may introduce a deadlock if they are not compatible with the application constraints. For instance, in the previous example, it should be checked that k is large enough so that A can trigger all r_A firings of B. This verification step is done using a deadlock detection algorithm presented in Sec. 3.3.

3.2 Resource Constraints

Resource constraints are used to regulate the parallel execution of actors. Such constraints can be used to limit the degree of parallelism or to enforce mutual exclusion between (groups of) actors. They can be seen as filter functions applied to the set of enabled (fireable) actors and returning a subset. Any such function f satisfies the two following conditions:

$$\forall \mathcal{S}.\ f(\mathcal{S}) = \mathcal{T} \Rightarrow \begin{cases} \mathcal{T} \subseteq \mathcal{S} & (C_1) \\ \mathcal{T} \neq \emptyset & (C_2) \end{cases}$$

Condition (C_1) ensures that the function is safe (only enabled actors can be selected), while (C_2) ensures that it preserves liveness (at least one actor is selected to be fired).

Many languages can be used to express such constraints. Since they are functions over finite domains, one may even consider expressing them exhaustively as tables. Here, we use rewrite rules on sets inspired from the Gamma formalism [1]. The general form of a resource constraint is:

$$\textbf{replace } S_A \textbf{ by } S_B \textbf{ if } \textit{condition} \tag{5}$$

where S_A and S_B are nonempty sets of enabled actors such that $S_B \subseteq S_A$. It can be read as "replace S_A by S_B if *condition* is

[1] This repetition vector assumes that r_A and r_B are coprimes.

true". When the *condition* is always true it can be omitted. For example, the rule

$$\textbf{replace}\ A,\ B\ \textbf{by}\ A \tag{6}$$

can be read as "if the actors A and B are in the set (of enabled actors), then replace them by A". It prevents actors A and B to be fired together and gives priority to A.

Rewrite rules can use pattern variables to match arbitrary actors. For instance, the rule

$$\textbf{replace}\ x,\ y,\ z\ \textbf{by}\ x, y \tag{7}$$

can be read as "select three arbitrary enabled actors and suppress one of them". It limits the level of parallelism to 2. Indeed, rewriting rules apply until no match can be found. Rule (7) above applies as long as there are more than two enabled actors.

Rules can also depend on a condition. For instance, assuming that the two predicates *short* and *long* denote whether an actor takes a short or long time to execute, the rule

$$\textbf{replace}\ x,\ y\ \textbf{by}\ x\ \textbf{if}\ \mathit{short}(x)\ \wedge\ \mathit{long}(y) \tag{8}$$

prevents short and long actors to be fired within the same slot (priority is given to short ones). This rule may improve the overall computation time. Indeed, if S is a "short" actor while L_1 and L_2 are two "long" actors such that S and L_1 are enabled at the same slot and firing S enables L_2, then it is better to fire first S alone and then L_1 and L_2 in parallel.

Several rules can also be combined in sequence or in parallel. The semantics of parallel composition enforces that rules applied in parallel act on disjoint sets of actors. Rules are applied repeatedly and terminate when no match can be found. For example, the sequential combination of rule (8) followed by rule (7) limits the possible parallel firings to one actor, two short actors, or two long actors. Additional examples of resource constraints are presented for the VC-1 decoder application in Sec. 4.

When actors are mapped on the same processing elements, resource constraints can be used to express their mutual exclusion. For instance, rule 6 can be used when actors A and B share the same processor. Although rule 6 gives priority to actor A, a condition can be added to express a more complex usage of the shared processor.

It is very easy to check that such rules preserve boundedness and liveness. If the *rhs* of Rule (5) is a non empty subset of its *lhs*, then the rule obeys conditions (C_1) and (C_2), and is, therefore, safe. For each application, they are statically compiled according to the set of actors into constant time selection operations.

3.3 Liveness analysis

User-defined ordering constraints may introduce deadlocks. For this reason, they must be checked statically for liveness. A set of ordering constraints may prevent liveness when they imply (by transitivity) a constraint of the form:

$$(A_i > A_j) \wedge (i \leq j) \tag{9}$$

which requires that the ith firing of an actor A must take place after the jth firing where j is a future firing ($j > i$). All cyclic constraints from an actor to itself must be checked. To ensure liveness, it must be shown that the deadlock condition in (9) is false for each cycle of the form:

$$A_i > B_{f_1(i)} > \cdots > C_{f_n(i)} > A_{f_{n+1}(i)}$$
$$\Rightarrow A_i > A_{f_1(\cdots(f_n(f_{n+1}(i))))}$$

hence that

$$i > f_1(\cdots(f_n(f_{n+1}(i)))) \tag{10}$$

We consider all ordering constraints to detect such cycles. Typically, the expression $f_1(\cdots(f_n(f_{n+1}(i))))$ contains parameters and ceiling functions. In general, only an upper bound can be computed. Parameters are replaced by their maximum or minimum values and ceilings $\lceil \frac{a}{b} \rceil$ by $\frac{a}{b}+1$ (or $\frac{a}{b}-1$) depending on their sign and position. The expression $f_1(\cdots(f_n(f_{n+1}(i))))$ can then be simplified to get an upper bound. If Equation (10) is true for all cycles, then the liveness of the schedule is guaranteed.

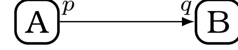

Figure 5. A simple BPDF graph

Consider, for instance, the simple BPDF graph in Fig. 5 where the user wants to limit the edge buffer to k tokens. In practice, such a limit (k) as well as the maximum values of parameters (p_{max}, q_{max}) are actual integers. Here, we illustrate the verification process using symbolic values. The graph constraint is:

$$B_i > A_{f(i)} \qquad \text{with} \quad f(i) = \left\lceil \frac{q \cdot i}{p} \right\rceil$$

and the user constraint that limits the buffer size to k is:

$$A_i > B_{g(i)} \qquad \text{with} \quad g(i) = \left\lceil \frac{p \cdot i - k}{q} \right\rceil$$

Together they form a cyclic constraint:

$$A_i > A_{f(g(i))}$$

To ensure liveness we must verify that:

$$\begin{aligned} i > f(g(i)) \quad &\Leftrightarrow \quad i > \left\lceil \frac{q \cdot \left\lceil \frac{p \cdot i - k}{q} \right\rceil}{p} \right\rceil \\ &\Leftarrow \quad i > \frac{q \cdot (\frac{p \cdot i - k}{q} + 1)}{p} + 1 \\ &\Leftarrow \quad i > i + \tfrac{q-k}{p} + 1 \\ &\Leftarrow \quad k > p + q \\ &\Leftarrow \quad k > p_{max} + q_{max} \end{aligned}$$

So, if the limit placed on the buffer size k is greater than $p_{max} + q_{max}$, a live schedule is ensured. This is onyl a ***sufficient*** condition, because of the approximation incurred by removing the ceiliing functions.

In general, if there exists a cycle that does not satisfy Equation (10), then user constraints involved are rejected. Actually, this cycle condition can be relaxed by taking boolean guards into account. Indeed, if the constraints occurring in the cycle depend on contradictory boolean guards, then the cycle is live as it cannot be formed.

3.4 Scheduler

When all the constraints are defined and checked for liveness, a scheduler is used to produce the slotted ASAP schedule for one iteration of the graph. The total number of firings of each actor must be equal to the solution of the balance equations. Two kinds of schedules can be distinguished:

Static schedules which are a finite sequence of actor instances, each repeated a constant number of times (1, 2, ...). For instance, $A(B|C)^2$ is a static schedule which starts by firing A then B and C in parallel twice (where '|' denotes parallel firings within one slot).

Quasi-static schedules which depend on the values of the integer or boolean parameters. For instance, $(A(b?\,B : C))^p$ is a quasi-static schedule that depends on both an integer and a

boolean parameter. Actor A is fired followed by B (resp. C) if b (resp. $\neg b$), and this sequence is iterated p times.

In general, the ASAP schedule of a BPDF graph can vary a lot in complexity, even when boolean parameters are absent. For example, the simple graph in Fig. 5 produces parametric constraints whose ceilings cannot be removed statically. The repetition vector is $[A^q\ B^p]$ and the ASAP scheduling of this simple graph must consider several cases:

Case $\mathbf{p \geq q}$: The slotted schedule is $A(A|B)^{q-1}B^{p-q+1}$. Indeed, once A has fired the first time, there are enough tokens to fire B at least once, because $p \geq q$. For each subsequent slot, A will fire in parallel with B until it has fired a total of $\#A = q$ times. This totals to $q-1$ firings of B so there remains to fire B another $p-(q-1)$ times.

Case $\mathbf{q > p}$: Then, two sub-cases must be considered:

- **Sub-case $\mathbf{q = k.p}$:** If q is a multiple of p, then the slotted schedule is $A(A^{k-1}(B|A))^{p-1}A^{k-1}B$, where each firing of B occurs after k firings of A. The total number of firings of A is $1+(k-1+1).(p-1)+k-1 = k.p = q$, while the total number of firings of B is $p-1+1 = p$.
- **Sub-case $\mathbf{q = k.p + r}$ with $\mathbf{0 < r < p}$:** Otherwise, the slotted schedule cannot be expressed by a regular formula as in the other cases. Indeed, it starts with the sequence $A^{k+1}B$, at which point there are $p-r$ tokens remaining in the edge. So, if $p \geq 2r$, then only k firings of A are necessary before B can be fired again, leaving $p-2r$ tokens; otherwise, A must be fired $k+1$ times before B can be fired, leaving $2p-2r$ tokens, and so on.

In general, we use a scheduler that evaluates the scheduling constraints at runtime. However, there are many cases where the scheduler can be simplified and implemented in a static or quasi-static way.

The scheduler takes as input the set of actors, the repetition vector, the writing/reading periods as well as the ordering and resource constraints. It processes the constraints and produces a schedule in a per slot manner (see Fig. 6). The scheduler stops when an iteration is finished and is reset to begin the next iteration.

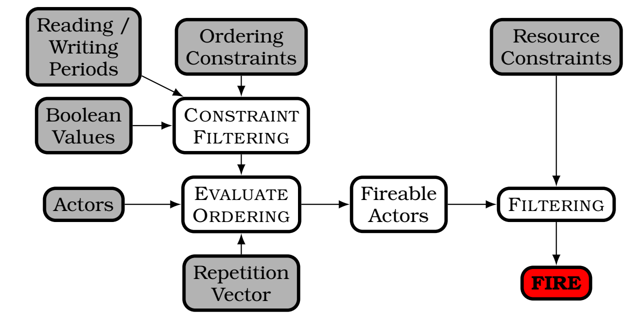

Figure 6. Scheduler overview

At the beginning of an iteration, the scheduler gets the values of integer parameters and possibly new boolean values if there are modifiers producing. Since all the reading periods are known, the scheduler can deduce which boolean value corresponds to which user firing (user firings may lag behind modifiers). Then, the scheduler filters out the set of ordering constraints based on the current values of the boolean parameters: data dependencies from disabled edges are not taken into account. Then, the set of remaining constraints is evaluated and a set of fireable actors is produced. Finally, the resource constraints evaluate the fireable set and select a subset of actors to fire. The scheduler reaches the end of the current iteration when the set of fireable actors is empty.

The scheduler is composed of three main functions: One function that filters out data dependencies based on the current boolean values (CONSTRAINT FILTERING), one function that evaluates the ordering constraints and produces a set of fireables actors (EVALUATE ORDERING), and finally one function that filters the set of fireable actors to a subset based on the resource constraints (FILTERING).

The filtering of the data dependencies (CONSTRAINT FILTERING) takes as input the set of ordering constraints ($\mathcal{C}$) and the values of the boolean parameters ($\mathcal{B}$). Then, it evaluates the boolean guards of each edge, and if the guard is false, the corresponding data dependency is removed from the set of ordering constraints. The procedure produces a reduced set of ordering constraints ($\mathcal{C}'$).

The evaluation of ordering constraints (procedure EVALUATE ORDERING in Fig. 7) takes as input a set of actors $\mathcal{A}$, a set of constraints $\mathcal{C}$, the repetition vector R and a status vector depicting the number of past firings per actor V_s. The output of the algorithm is the fireable vector V_f that flags the fireable actors for the current slot.

procedure EVALUATE ORDERING($R, \mathcal{C}, \mathcal{A}, V_s$)
  $V_f \leftarrow \vec{0}$
  for $\forall X \in \mathcal{A}$ **do**
    if $(\text{EVAL}(\mathcal{C}(X), V_s)) \wedge (V_s[X] < R[X])$ **then**
      $V_f[X] \leftarrow 1$
    end if
  end for
  return(V_f)
end procedure

Figure 7. Evaluation of ordering constraints.

We denote $R[X]$ the number of firings of actor X required by the iteration and $\mathcal{C}(X)$ the set of constraints imposed on X (*i.e.,* all constraints of the form $X_i > \ldots$).

The core of the algorithm is the evaluation of constraints represented by the function EVAL, which evaluates the constraints of an actor ($\mathcal{C}(A)$) according to the current status vector (V_s). More precisely, for each constraint

$$X_i > Y_{f(i)}$$

the EVAL function simply checks whether:

$$f(V_s[X]+1) \leq V_s[Y]$$

which corresponds to the satisfaction of the data dependency. Indeed, $V_s[X]+1$ represents the index of the next firing of X and $f(V_s[X]+1)$ represents the number of firings that actor Y should have achieved before we can fire $X_{V_s(X)+1}$ due to the $X_i > Y_{f(i)}$ constraint. If the current number of firings of Y (*i.e.,* $V_s[Y]$) is greater than the index of the next firing of X (*i.e.,* $f(V_s[X]+1)$) then the constraint

$$X_i > Y_{f(i)} \text{ with } i = V_s[X]+1$$

is satisfied. If all the constraints on X are satisfied, then EVAL returns true and X is allowed to be fired in the next slot. Otherwise, it returns false and X will not be fired.

Apart from the ordering constraints, the repetition vector is also checked, $(V_s[X] < R[X])$, to determine whether the actor needs to be fired again in the current iteration. If both conditions are satisfied, $V_f[X]$ is set to 1. After all actors have been considered, the fireable vector is produced. The deadlock detection algorithm ensures that the inner loop always terminates.

Since each actor is selected as soon as its constraints are met, the procedure produces an ASAP schedule *w.r.t.* constraints. We choose ASAP scheduling because it produces highly parallel schedules. Actually, without timing information, it can be shown to be the most parallel slotted schedule. Moreover, it ensures a minimal schedule length in terms of number of slots.

When the fireable vector has been produced, it is used as input, along with the resource constraint matrix $\mathcal{G}$ by the FILTERING procedure. FILTERING is just a lookup procedure that finds V_f in the constraint table and returns the entry of the table for that vector, so we do not provide its pseudo code. The output of FILTERING is a new firing vector $V'_f \subseteq V_f$ containing all the actors to be fired in the current slot that also updates the status vector V_s.

Our scheduler is summarized in Fig. 8. The inputs are the set of actors $\mathcal{A}$, the two sets of constraints ordering and resource $(\mathcal{C}, \mathcal{G})$ and the repetition vector R.

procedure SCHEDULER($\mathcal{A}, \mathcal{C}, \mathcal{G}, R$)
  while true **do**
    READ INTEGER VALUES()
    $V_s \leftarrow \vec{0}$
    $F \leftarrow \vec{0}$
    while $V_s \neq R$ **do**
      $\mathcal{B} \leftarrow$ READ BOOLEAN VALUES()
      $C' \leftarrow$ CONSTRAINT FILTERING($\mathcal{C}, \mathcal{B}$)
      $V_f \leftarrow$ EVALUATE ORDERING($R, \mathcal{C}', \mathcal{A}, V_s$)
      $V'_f \leftarrow$ FILTERING($V_f, \mathcal{G}$)
      $V_s \leftarrow V_s + V'_f$
      fire(F)
    end while
  end while
end procedure

Figure 8. Scheduler algorithm.

The SCHEDULER procedure is structured as an outer infinite loop and an inner loop that iterates over the iteration of the graph. As both functions, EVALUATE ORDERING and FILTERING, guarantee to fire at least one actor as long as the the repetition vector is not reached, the inner loop terminates when the iteration is complete. The outer loop resets the auxiliary vectors and repeats the iteration.

Actors are scheduled and fired one slot at a time. While actors execute, the SCHEDULER procedure concurrently evaluates constraints to find the actors that must be fired at the next slot.

The slotted scheduling model may introduce a lot of slack in the produced schedule because of the explicit synchronization after every slot. This is inherent to the model but can be mitigated using constraints to group actors with similar timings in slots (see Sec. 4).

The slotted model was prescribed by our target platform but we should point out that our framework can also be used to produce non slotted schedules. In our context, where each actor is a separate processing element, the ASAP non-slotted schedule is optimal *w.r.t.* to time and constraints. The scheduler main-loop needs to be adjusted so that the status vector is updated each time an actor ends its firing (instead of at the end of each slot). The scheduler re-evaluates constraints each time an actor ends, finds new enabled actors and fires them. An extra vector recording the active (*i.e.,* currently executing) actors is also needed. It is used to prevent executing actors to be considered during constraint evaluation and also to evaluate resource constraints which now apply on enabled *and* already active actors.

3.5 Constraint simplification

Scheduling can be optimized in several cases. For an SDF graph (*e.g.,* without parameters), all constraints can be solved statically by considering the constraints for each individual firing (thus getting rid of the index i). Then, the scheduling algorithm boils down to a sequence of firings.

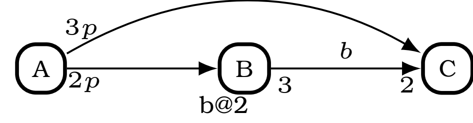

Figure 9. BPDF graph with constraints that can be solved symbolically

Even for parametric graphs, it is often possible to solve ordering constraints symbolically. Consider the graph in Fig. 9, whose iteration is $AB^{2p}C^{3p}$ and whose dataflow constraints are:

$$B_i > A_{\lceil \frac{i}{2p} \rceil}, \;\; C_i > B_{\lceil \frac{2i}{3} \rceil}, \;\; C_i > A_{\lceil \frac{i}{3p} \rceil}$$

plus the implicit $X_i > X_{i-1}$ for all actors. Moreover, actor C, as a user of the boolean parameter b, is constrained by (from Eq. 3):

$$C_i > B_{2\lfloor \frac{i-1}{3} \rfloor + 1} \;\; \Rightarrow \;\; C_i > B_{2\lceil \frac{i}{3} \rceil - 1}$$

Knowing that there is only one firing of A in each iteration, and that A is unconstrained, we schedule A in the first slot and we get $A_1 = 1$. For actor B, the constraint becomes:

$$B_i > A_1 \;\; \Rightarrow \;\; B_i > 1$$

Since B_i should be fired as soon as the constraints are satisfied, its constraints are rewritten into the equation:

$$B_i = max(B_{i-1}, 1) \;\; \text{for } i \in [1..2p]$$

which can be solved to $B_i = i + 1$ indicating that the ith firing of B will fire in the $i + 1$th slot. Finally, for actor C we have three constraints:

$$C_i > A_{\lceil \frac{i}{3p} \rceil} \;\; \Rightarrow \;\; C_i > 1$$

$$C_i > B_{\lceil \frac{2i}{3} \rceil} \tag{11}$$

$$C_i > B_{2\lceil \frac{i}{3} \rceil - 1} \tag{12}$$

which form the equation:

$$C_i = max(A_1, B_{\lceil \frac{2i}{3} \rceil}, B_{2\lceil \frac{i}{3} \rceil - 1}, C_{i-1}) \;\; \text{for } i \in [1..3p] \tag{13}$$

The two constraints on actor B dominate the one on actor A and the data dependency (11) dominates over the modifier - user dependency (12) as $\lceil \frac{2i}{3} \rceil \geq 2\lceil \frac{i}{3} \rceil - 1$. So, Equation (13) yields:

$$C_i = max(B_{\lceil \frac{2i}{3} \rceil}, C_{i-1}) + 1 \;\; \text{for } i \in [1..3p]$$

However, when the boolean parameter b is set to $false$, the data dependency is not taken into account and Equation (13) yields:

$$C_i = max(B_{2\lceil \frac{i}{3} \rceil - 1}, C_{i-1}) + 1 \;\; \text{for } i \in [1..3p]$$

In both cases, the solution for actor C is found to be $C_i = i + 2$, so the value of the boolean parameter does not influence the schedule. The resulting schedule can be expressed as regular expressions:

$$A \; B \; (B|C)^{2p-1} \; C^{p+1}$$

Such schedules can be implemented as standard quasi-static schedules. However, in general, resource constraints and boolean parameters entail some dynamic checks in the scheduler.

4. Case studies

4.1 The VC-1 decoder

The VC-1 decoder [14] is a good example of a demanding codec. Its resemblance with the more recent and widely used H.264 [14] and with future generation codecs like HEVC, makes it especially relevant. The BPDF implementation of VC-1 is shown in Fig. 10.

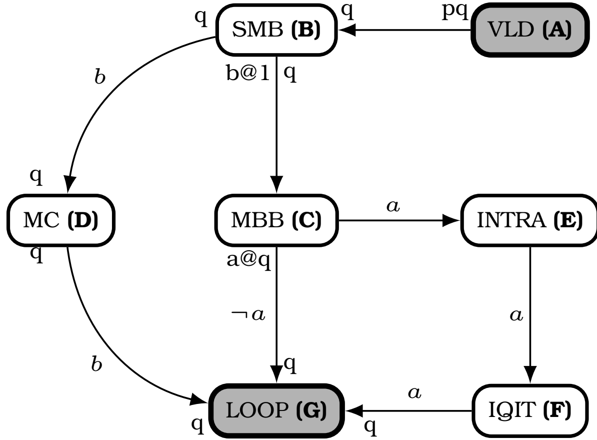

Figure 10. BPDF capture of VC-1 decoder

The decoder is composed of two main pipelines, the inter and the intra. The inter pipeline is composed of actor MC (Motion Compensation), while the intra pipeline is composed of actors MBB (MacroBlock to Block), INTRA (Intra prediction), and IQIT (Inverse Quantization and Inverse Transform). These two paths are combined and produce the final decoded slice in the LOOP (Loop filter). Actors SMB (Slice to MacroBlock) and MBB are auxiliary actors that are used as modifiers of the boolean parameters. For easier reference, each actor is assigned a letter (shown in "()").

The inter pipeline reconstructs data based on motion between different frames. For this, it fetches data from previous or future frames and, based on motion vectors, compensates the motion for the current macroblock. The intra pipeline reconstructs the data that depends on macroblocks in the neighborhood of the decoding macroblock. The intra prediction actor calculates coefficients based on this information, the IQIT applies inverse transformations to complete the decoding of the data. Finally, the residues of both pipelines are combined and smoothed in the loop filter.

The decoder makes use of two integers and two boolean parameters. The integer parameters are p, which encodes the slice size in macroblocks, and q, which encodes the macroblock size in blocks. Each iteration of the graph processes a single slice. The boolean parameters capture whether a block is using intra (a) or inter (b) information. With these two boolean parameters, three possible modes of operation can be distinguished:

$$a \wedge \neg b : \quad \text{Intra only}$$
$$\neg a \wedge b : \quad \text{Inter only}$$
$$a \wedge b : \quad \text{Intra and Inter}$$

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.

By solving the balance equations, we get the repetition vector

$$[AB^pC^{pq}D^pE^{pq}F^{pq}G^p]$$

The graph is first scheduled, with no additional constraints, as explained in the previous section.

Actor	Execution Time (Cycles/Firing)	ASAP Sequences
VLD (A)	7400	$\mathcal{F}$
SMB (B)	10	$\mathcal{E}\mathcal{F}^p$
MBB (C)	10	$\mathcal{E}^2\mathcal{F}^{pq}$
MC (D)	1937	$\mathcal{E}^2\mathcal{F}^p$
INTRA (E)	288	$\mathcal{E}^3\mathcal{F}^{pq}$
IQIT (F)	365	$\mathcal{E}^3\mathcal{R}_F(pq)$
LOOP (G)	4074	$\mathcal{E}^3\mathcal{R}_G(p)$

Table 1. ASAP sequences of VC-1

The resulting schedule cannot be expressed as a single sequence using the notation introduced in Section 3.4. It is possible though, to express the schedule using individual execution sequences for each actor as shown in Table 1 (third column). Each one represents the sequence of slots of the iteration where either the actor is fired (written $\mathcal{F}$) or it remains idle (written $\mathcal{E}$). The possible idle slots after the last firing of actors are omitted.

In the case of actors IQIT and LOOP, the schedule is depending on the boolean value of a and shows increased dynamicity. To express the sequence of firings, we use two recursive functions $\mathcal{R}_F$, for IQIT and $\mathcal{R}_G$ for LOOP defined as follows:

$$\mathcal{R}_F(n) = a\ ?\ \mathcal{E}\ \mathcal{F}^n\ :\ \mathcal{F}^q\ \mathcal{R}_F(n-q)$$

$$\mathcal{R}_G(n) = a\ ?\ \mathcal{E}^{q+1}\mathcal{F}\ \mathcal{R}_T(n-1)\ :\ \mathcal{E}^{q-1}\mathcal{F}\ \ \mathcal{R}_G(n-1)$$
$$\mathcal{R}_T(n) = a\ ?\ \mathcal{E}^{q-1}\mathcal{F}\ \mathcal{R}_T(n-1)\ :\ \mathcal{E}^{q-3}\mathcal{F}\ \mathcal{R}_F(n-1)$$
$$\mathcal{R}_F(n) = a\ ?\ \mathcal{E}^{q+1}\mathcal{F}\ \mathcal{R}_T(n-1)\ :\ \mathcal{E}^{q-1}\mathcal{F}\ \mathcal{R}_F(n-1)$$

The sequence associated to IQIT (F) means that IQIT remains idle in the first 3 slots, and then if a is $true$, it waits one more slot and fires consecutively until he finishes its iteration. If a is $false$, IQIT fires for q slots and then checks again the value of a.

The execution sequence of LOOP (G) is a more complex one, as it depends not only on the boolean values but also on their sequence. For this reason, the functions $\mathcal{R}_T$ and $\mathcal{R}_F$ are used for when the boolean is $true$ and $false$ respectively. The complete schedule is the parallel combination of all execution sequences. It exhibits a high level of parallelism and a sample execution starts as

$$(A)\ (B)\ (B|C|D)\ (B|C|D|E)\ (B|C|D|E|F)\ \ldots$$

The total span of the produced schedule has a maximum of $pq + 5$ slots and a minimum of $pq + 3$ slots. By adding user-constraints, we can modify the ASAP schedule to improve it or satisfy some given criteria. In the following, examples of ordering and resource constraints are given. To evaluate the decoder's performance we reused the VC-1 performance on STHORM based on the implementation presented in [2]. The execution time of each actor's firing is shown in Table 1 (second column).

4.2 Ordering constraint examples

The inter-prediction path processes one macroblock at a time whereas the intra-prediction path processes one block at a time. Consequently, actors in the intra-prediction are fired a total number

of pq times, whereas D fires only p times. This results into D firing in the early slots and producing a lot of tokens on the edge (D, G). However, G cannot consume these tokens because it is blocked by the intra-prediction pipeline.

Using additional constraints, we can limit the buffer size of the edge (D, G) and prevent the accumulation of data in the inter-prediction path. To produce the alternative schedule, we delay the inter-prediction path by constraining the (D) actor to wait until (G) has consumed q tokens. Using the constraint from (4) we get:

$$D_i > G_{\lceil \frac{q \cdot i - q}{q} \rceil} \Rightarrow D_i > G_{i-1} \qquad (14)$$

The constraint adds idle intervals of $q - 1$ slots between the firings of D. This redistribution of the firings of D has the additional benefit of a more evenly distributed power consumption, and subsequently a smaller temperature. Although the schedule span of actor D increases, we observed only a slight increase of 2% to the total schedule time, so the total schedule span effectively remains the same.

This significant change on the graph schedule is achieved by adding a single constraint. It demonstrates the flexibility of our scheduling framework.

4.3 Resource constraint examples

When slotted scheduling is used, the goal is to minimize the introduced slack because of the synchronization after each slot. For this reason, we try to cluster together the more cycle-demanding actors. In Table 1, we notice that, apart from A that fires only once, the most costly actors are D and G. An obvious optimization is to fire them in the same slots.

We can use a resource constraint to achieve this goal. By looking at the actors' schedule streams of Table 1, we see that all firings of D, after the first one, are fired in parallel with E. The following constraint can be used:

$$\textbf{replace}\ D,\ E\ \textbf{by}\ E\ \ \textbf{if}\ \neg\texttt{fireable}(G) \qquad (15)$$

This constraint suppresses D when G is not present, effectively clustering the two actors together. This extra constraint led to an improvement of 15% in the total schedule time. A non-slotted schedule (optimal *w.r.t.* timing in our context) would improve the total execution time by an additional 30%.

Resource constraints can also be used to restrict concurrency and power consumption of the VC-1 application. For instance, if we want to limit its parallelism to at most 3 concurrent actors, we can use the following resource constraint:

$$\textbf{replace}\ w, x, y, z\ \textbf{by}\ w, x, y \qquad (16)$$

We may additionally want to limit the power consumption of the chip during a slot. Assuming two predicates that classify actors into high (H) or low (L) power consumers, we can limit power consumption by firing at most one H actor either alone or along with at most one L actor. The following set of rules implements such a limitation:

$$\begin{array}{lll} \textbf{replace}\ x,\ y & \textbf{by}\ x & \textbf{if}\ H(x) \wedge H(y) \\ \textbf{replace}\ x,\ y,\ z & \textbf{by}\ x,\ y & \textbf{if}\ H(x) \wedge L(y) \wedge L(z) \end{array}$$

With precise information about actors (that we do not currently have), total power consumption could be better controlled, *e.g.,* bounded by a specified limit). Further experimentation is needed to demonstrate the way VC-1's schedule can be altered and optimized using the above constraints, however the platform is not readily available to us yet.

4.4 Scheduler overhead evaluation

Temporal Noise Reduction (TNR) is an algorithm applied after the video decoding process to reduce the noise of each frame. We implemented TNR on the STHORM platform using BPDF and used the scheduling framework to schedule it. The BPDF graph of TNR processes one frame per iteration. We measured the average cycles used by the processor that schedules the graph for each frame.

We consider three different cases: the original manual implementation of the scheduler for TNR (*Manual Sched.*), our BPDF scheduler without any simplification of constraints(*BPDF no opt.*) and our BPDF quasi-static schedule produced after simplification of constraints (*BPDF opt.*). Table 2 shows the number of cycles taken by the scheduler at each frame (line 2) and the maximum QoS (in terms of fps) the scheduler or actors could meet (line 3). For comparison, column 2 shows the corresponding numbers for the faster actor of TNR. We can see that the dynamic scheduler

	Best Actor Performance	**BDPF no opt.**	**BPDF opt.**	**Manual Sched.**
Cycles / frame	2.140.000	1.100.000	360.000	340.000
FPS	187	363	1111	1176

Table 2. Schedule overhead for different schedules of TNR

introduces a large overhead (almost three times more costly than the manual schedule). Once the constraints are simplified and the schedule is reduced to a quasi-static one, the overhead is comparable with that of the manual schedule. For VC-1, the required quality of service is 30 Frames/sec and in the best case an actor is much more costly than the scheduler. So, in the best case, although the unoptimized scheduler introduces large overhead, since it runs in parallel with such coarse grain actors, the required QoS is still achieved. In that context, a dynamic scheduler is realistic and allows the use of additional constraints to optimize various criteria.

5. Related Work

Parallel scheduling of data flow graphs is an old problem that has been dealt with for many years. In the case of SDF [13], it typically involves the transformation of the graph to a homogeneous SDF (HSDF) format where all actors produce or consume a single token [12]. It exposes parallelism and allows to use popular techniques like list scheduling. The hierarchical scheduling framework proposed in [18] uses a clustering technique to prevent the actor explosion that occurs when SDF is transformed into HSDF. These approaches apply only to SDF graphs however, and although there is a suggestion of a scheduler to optimize the schedule, it has never been explored.

SDF graphs are fully compatible with our framework. In the case of a SDF graph, all constraints can be resolved statically. Scheduling can be resolved at compile time and produce a static ASAP parallel schedule for the SDF graph. Different scheduling strategies can be expressed using specific user-defined constraints. Such constraints can be checked and integrated in the static schedule.

When the expressiveness of data flow models increases, so does the complexity of their scheduling. To deal with switch actors and conditional execution, Lee proposed quasi-static schedules [11] expressed with iterations and conditionals which must be evaluated at run-time. A similar approach has been explored further by Ha & Lee [9], where quasi-static schedules for data-dependent data flow graphs are produced. They consider schedules that have the same frontier regardless the presence of switch actors (expressing

conditional execution) or actors with an iteration based on the data. In our case though, we explore the production of self-timed schedules using a different scheduling model based on slots.

Bhattacharya & Bhattacharyya [5] explore the use of quasi-static scheduling to produce schedules for parameterized dataflow applications captured in the PSDF model [6]. A clustering technique is used to produce parameterized looped schedules. In [19] the approach is generalized to schedules expressed using generalized schedule trees. However, these schedules are sequential.

In [10], the use of generalized scheduling trees along with an analysis that minimizes buffers is used to produce quasi-static schedules for parameterized cyclo-static dataflow graphs [7]. The expressiveness of the approach is reduced because of restrictions of the clustering mechanism and limitations of the schedule representation. Finally, there is no flexibility to alter the schedule nor any liveness guarantees. Our framework is compatible with these scheduling techniques as it can express such parameterized looped schedules using the appropriate constraints.

6. Conclusions

We have presented a framework to specify and implement bounded, live and highly parallel schedules for boolean parametric dataflow graphs. Scheduling is made flexible by the use of user constraints that allow the framework to adapt to new execution platforms, express optimizations and regulate parallel firings. Static checks can ensure that constraints preserve the existence of bounded and live schedules.

The approach was used to schedule two streaming applications (VC-1 and TNR) on the STHORM platform demonstrating the feasibility and flexibility of the approach. The framework facilitates the automatic production of complex schedules that can be as efficient as the manual ones, which are often hard to produce and error-prone.

The main aim of our framework is not to produce the optimal quasi-static schedule but to propose a flexible and correct by construction approach that can easily express different schedules and scheduling strategies. The framework can then be used to explore the various scheduling possibilities and to optimize the schedule *w.r.t.* various criteria.

Although we only considered BPDF, our framework can be adapted to schedule other data flow models as long as their data flow constraints can be expressed in our constraint language. We believe that the framework can also accommodate other models of computation such as Petri nets and process networks. Similarly, if the framework was designed with slotted scheduling in mind, it can easily be converted to non-slotted models without compromising its flexibility.

As future work, we have the following mid-term objectives:

- Make use of constraints to optimize bi-criteria scheduling, specifically power consumption vs. throughput;
- Design a high-level language to express scheduling policies that can be automatically compiled into constraints, which can in turn be taken into account by the scheduler.

References

[1] J.-P. Banâtre and D. Le Métayer. Programming by multiset transformation. *Comm. of the ACM*, 36(1):98–111, Jan. 1993.

[2] M. Bariani, P. Lambruschini, and M. Raggio. Vc-1 decoder on stmicroelectronics p2012 architecture. In *Proc. of 8th Annual Intl. Workshop 'STreaming Day'*, Sept 2010. .

[3] V. Bebelis, P. Fradet, A. Girault, and B. Lavigueur. BPDF: A statically analyzable dataflow model with integer and boolean parameters. In *ACM Int. Conf. Embedded Software, EMSOFT'13*, pages 1–10, Montreal, Canada, Sept. 2013.

[4] L. Benini, E. Flamand, D. Fuin, and D. Melpignano. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *Design Automation and Test in Europe, DATE'12*, pages 983–987, 2012.

[5] B. Bhattacharya and S. S. Bhattacharyya. Quasi-static scheduling of reconfigurable dataflow graphs for DSP systems. In *IEEE International Workshop on Rapid System Prototyping*, pages 84–89, 2000.

[6] B. Bhattacharya and S. S. Bhattacharyya. Parameterized dataflow modeling for DSP systems. *IEEE Trans. on Signal Processing*, 49 (10):2408–2421, 2001.

[7] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static dataflow. *IEEE Trans. on Signal Processing*, 44(2):397–408, 1996.

[8] P. Fradet, A. Girault, and P. Poplavko. SPDF: A schedulable parametric data-flow MoC. In *Design Automation and Test in Europe, DATE'12*, pages 769–774, 2012.

[9] S. Ha and E. A. Lee. Compile-time scheduling and assignment of data-flow program graphs with data-dependent iteration. *IEEE Trans. Computers*, 40(11):1225–1238, 1991.

[10] H. Kee, C.-C. Shen, S. S. Bhattacharyya, I. Wong, Y. Rao, and J. Kornerup. Mapping parameterized cyclo-static dataflow graphs onto configurable hardware. *Signal Processing Systems*, 66(3):285–301, 2012.

[11] E. A. Lee. Recurrences, iteration, and conditionals in statically scheduled block diagrams languages. In *VLSI Signal Processing III*, chapter 31, pages 330–340. IEEE Press, 1988.

[12] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Computers*, 36(1):24–35, 1987.

[13] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *IEEE Trans. Computers*, 36(1):24–35, 1987.

[14] J.-B. Lee and H. Kalva. *The VC-1 and H.264 Video Compression Standards for Broadband Video Services*. Springer, 2008.

[15] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.

[16] A. Munshi. *The OpenCL Specification*. Khronos OpenCL Working Group, 1.1 edition, June 2011.

[17] *NVIDIA CUDA Programming Guide*. NVIDIA Corp., 4.1 edition, 2012.

[18] J. L. Pino, S. S. Bhattacharyya, and E. A. Lee. A hierarchical multiprocessor scheduling framework for synchronous dataflow graphs. Tech. report UCB/ERL M95/36, Univ. of California at Berkeley, May 1995.

[19] W. Plishker, N. Sane, and S. S. Bhattacharyya. A generalized scheduling approach for dynamic dataflow applications. In *Design Automation and Test in Europe, DATE'09*, pages 111–116, Nice, France, Apr. 2009.

[20] S. Sriram and S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., New York, NY, USA, 1st edition, 2000. ISBN 0824793188.

[21] 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 *International Conference on Formal Methods and Models for Codesign, MEMOCODE'06*, pages 185–194, Napa Valley (CA), USA, July 2006. ACM-IEEE.

[22] 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.