Symbolic Buffer Sizing for Throughput-Optimal Scheduling of Dataflow Graphs

Adnan Bouakaz Pascal Fradet Alain Girault

INRIA; Univ. Grenoble Alpes
first.last@inria.fr

***Abstract*—The synchronous dataflow model is widely used to design real-time streaming applications which must assure a minimum quality-of-service. A benefit of that model is to allow static analyses to predict and guarantee timing (*e.g.,* throughput) and buffering requirements of an application. Performance analyses can either be performed at compile time (for design space exploration) or at run-time (for resource management and reconfigurable systems). However, these algorithms, which often have an exponential time complexity, may cause a huge run-time overhead or make design space exploration unacceptably slow. In this paper, we argue that symbolic analyses are more appropriate since they express the system performance as a function of parameters (*i.e.,* input and output rates, execution times). Such functions can be quickly evaluated for each different configuration or checked *w.r.t.* many different non-functional requirements. We first provide a symbolic expression of the maximal throughput of acyclic synchronous dataflow graphs. We then perform an analytic and exact study of the minimum buffer sizes needed to achieve this maximal throughput for a single parametric edge graph. Based on these investigations, we define symbolic analyses that approximate the minimum buffer sizes needed to achieve maximal throughput for acyclic graphs. We assess the proposed analyses experimentally on both synthetic and real benchmarks.**

I. INTRODUCTION

Synchronous dataflow (SDF) graphs [11] are widely used to design digital signal processing and concurrent real-time streaming applications. This model comes with static analyses that guarantee the *boundedness* and *liveness* of an application as well as *predictable performances* (e.g. throughput, latency, memory requirements).

Performance analyses of SDF graphs check whether non-functional requirements are met. They can be performed both at design time and at run-time. At design time, it is a crucial step in the development of embedded applications. Many decisions and settings of the system need to be explored (e.g. hardware/software partitioning, memory allocation, granularity and different implementations of tasks, processor speeds, etc.) and the best options that satisfy the non-functional requirements can be chosen. At run-time, performance analysis is performed either for resource management or to cope with the dynamic behavior of parametric versions of SDF. Indeed, in response to the increasing complexity of systems, many parametric dataflow models have been proposed (*e.g.,* PSDF [4], SPDF [9], BPDF [2], πSDF [8], SADF [7], *etc.*) in which the graph (*e.g.,* its rates or channels) may change at run-time.

Throughput is one important timing constraint of real-time systems. For example, a video decoder should decode a minimum number of frames per second. Using a throughput-optimal scheduling policy, such as self-timed scheduling, allows the designer to guarantee such timing requirements, but also to use dynamic voltage and frequency scaling (DVFS) techniques to reduce energy consumption in case the maximum throughput is larger than the desired quality-of-service [15]. Furthermore, for embedded systems, it is primordial to determine the minimum buffer sizes that allow such scheduling.

In this paper, we propose *symbolic* analyses of dataflow graphs. Our analyses consider the rates and execution times of actors as parameters. Most non-functional properties of the application can be described as a function of these parameters. By evaluating these functions for specific values of parameters, the properties/performances of specific configurations can be obtained. We propose two symbolic analyses of acyclic graphs *under self-timed scheduling* to answer the following questions:

Q1. What is the *maximum throughput* of the application?
Q2. What are the *minimum buffer sizes* that allow the application to achieve its maximum throughput?

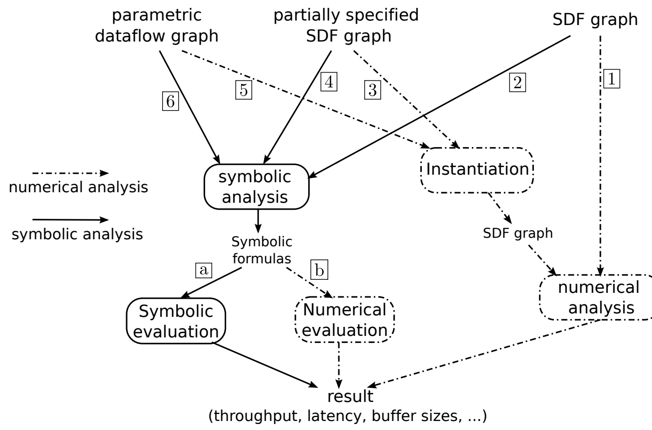


Fig. 1: Symbolic and numerical analyses.

Although our symbolic analyses may give only approximate (but safe) results, they are quite useful in many cases (see Fig. 1) :

1. At early design stages, the SDF model of the application is usually partially specified and many system settings remain to be explored. For instance, the execution time of an actor depends on whether it is implemented in software or in hardware, or which algorithm will be used to implement its functionality, *etc.* Therefore, design space exploration may require analyzing of a potentially huge number of configurations (path $\boxed{3}$). Symbolic analyses are a big advantage in this case. Formulas are generated only once and simply evaluated for each possible configuration (*i.e.,* set of parameters) (path $\boxed{4}$).

2. Similarly, non-functional requirements of parametric dataflow models *at compile time* can be expressed symbolically as parametric formulas. Then, the requirements can be either checked by *evaluating* formulas for all potential configurations (path $\boxed{6b}$) or, better, by an *analytic proof* (path $\boxed{6a}$) to ensure, for instance, that the buffering requirements are less than some threshold whatever the configuration of the application.

3. For dynamic models and runtime resource management, appropriate settings have sometimes to be chosen dynamically. Consider, for instance, a parametric application where frequency scaling is used to guarantee a specific throughput and minimize power consumption. In such case, frequency must be adjusted at each parameter change. Instantiating the graph (path $\boxed{5}$) and performing a numerical analysis is far too costly at run time. Consequently, fast analyses, like the evaluation of symbolic formulas, are required (path $\boxed{6b}$).

4. Finally, even for completely static SDF models, many analyses have an exponential complexity. In practice, the exact computation of throughput and latency is acceptable at compile time (path $\boxed{1}$). However, exact algorithms for minimal buffer sizes are too expensive even for small graphs. Moreira *et al.* [12] shows that this problem is NP-complete for homogeneous SDF (HSDF) graphs. Moreover, SDF-to-HSDF conversion may lead to an *exponential* growth of the size of the graph. Our symbolic analysis (path $\boxed{2}$) is much more efficient and its approximate solution can also be considered as a starting point to prune the parameter space and hence improve the performance of the exact algorithm.

Contributions. The contributions of our paper are threefold:

a. We present and prove a duality theorem and show how it can be used to prune many cases.

b. We give a *new analytic* characterization of the parametric data-dependency $A \xrightarrow{p \quad q} B$, called enabling patterns, which can be used to build symbolic analyses, *e.g.,* for buffer sizing.

c. We describe exact and approximate *polynomial-time* symbolic analyses to answer the above questions (**Q1** and **Q2**).

The article is organized as follows. Section II introduces synchronous dataflow graphs and the needed definitions. Section III presents throughput analysis and a new generic result required to solve the second question. We present in Section IV an exact analysis of the minimum buffer size for a single edge SDF graph $A \xrightarrow{p \quad q} B$. These results are extended to acyclic graphs using approximate analyses in Section V. We evaluate our technique on both synthetic and real benchmarks in Section VI. Finally, we review related work in Section VII and conclude in Section VIII. The interested reader will find proofs and additional details in a companion paper [6].

II. Background

In this section, we first introduce the application model and the scheduling policy. Then, we review some useful properties.

Application model: An SDF graph $G = (V, E)$ consists of a finite set of *actors* V and a finite set of *edges* E which can be seen as FIFO channels. The atomic execution of a given actor (called *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 (resp. produced) at a given port at each firing is called the consumption (resp. production) *rate*. An actor can fire only when all its input edges have enough tokens (*i.e.,* at least the number specified by the corresponding rate). An edge may contain some initial tokens (also called delays). The execution time of an actor X is denoted by t_X. For instance, Fig. 2(a) shows an SDF graph with two actors A and B, with execution times $t_A = 20$ and $t_B = 7$ respectively. The production and consumption rates are 8 and 5 respectively. The top edge in Fig. 2(b) has 15 initial tokens (represented by the black dot).

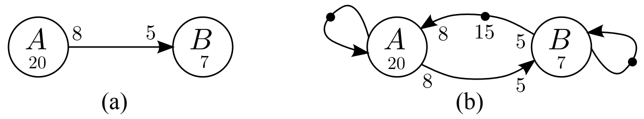


Fig. 2: (a) An SDF graph; (b) The same graph with channel size constraint and auto-concurrency disabled.

The *state* of a dataflow graph is the vector of number of tokens present at each edge (*i.e.,* buffered in each FIFO channel). Each edge carries zero or more tokens at any moment. The *initial state* of the graph is specified by the number of initial tokens. The initial state of the graph of Fig. 2(b) is $[i_{AA}{=}1, i_{AB}{=}0, i_{BB}{=}1, i_{BA}{=}15]$. An iteration of an SDF graph is a non empty sequence of firings that returns the graph to its initial state. For the graph in Fig. 2(a), firing actor A five times and actor B eight times forms an iteration. The repetition vector $\vec{z} = [z_A{=}5, z_B{=}8]$ indicates the number of firings of actors per iteration. If such vector exists the graph is said to be *consistent* [11]. We denote by z_X the number of firings of actor X in the iteration.

Homogeneous SDF (HSDF) is a restriction of SDF where all the production and consumption rates are equal to 1. HSDF graphs are particularly useful because (*i*) their throughput can be computed as the inverse of the *maximal cycle mean* (MCM); and, (*ii*) any SDF graph can be converted into an equivalent HSDF graph. The cycle mean of a cycle is equal to the sum of execution times of the actors in the cycle divided by the number of delays (*i.e.,* initial tokens) in the channels of this cycle. This provides a way to compute the throughput of any SDF graph. Unfortunately, the translation from SDF to HSDF may lead to an exponential increase of the number of nodes.

Scheduling policy: In this paper, we focus on as soon as possible (ASAP) scheduling of consistent graphs without auto-concurrency (*i.e.,* two firings of the same actor cannot overlap). In such self-timed executions [14], an actor fires as soon as if it is idle (no auto-concurrency) and has enough tokens on its input channels. We assume there are sufficient processing units, *e.g.,* there are as many processors as actors or actors are implemented in hardware. ASAP scheduling allows to reach the maximal throughput. Such schedules are naturally pipelined and composed of a prologue followed by a *steady state* that repeats infinitely. Fig. 6 shows the ASAP schedule of the graph in Fig. 2(a).

The multi-iteration latency of the first n iterations of the execution of a graph G, written $\mathcal{L}_G(n)$, is equal to the finish time of the last firing in all firings composing the first n iterations (assuming timing starts at the very first firing). The period of the execution (denoted by $\mathcal{P}_G$) is the average length of an iteration and formally defined as

$$\mathcal{P}_G = \lim_{n \to \infty} \frac{\mathcal{L}_G(n)}{n}$$

The throughput of the execution, $\mathcal{T}_G$, is the number of iterations per unit of time, and hence equals $\frac{1}{\mathcal{P}_G}$.

The FIFO channel $A \xrightarrow{p \quad q} B$ with bounded size d can be modeled by adding a backward channel $B \xrightarrow{q \quad p} A$ with d initial tokens, as shown in Fig. 2(b). This conservative modeling, assumed in most works, enforces that an actor can start firing only if there is enough space on its output channels. Fig. 2(b) also shows how to make the prevention of auto-concurrency explicit by adding self-edges with rates equal to 1 and one initial token.

For a channel $A \xrightarrow{p \quad q} B$ with d initial tokens, the i^{th} firing of B (denoted B_i) is enabled if and only if the number of produced tokens is larger than iq. Hence, B has to wait for the j^{th} firing of A (denoted A_j) such that $jp + d \geq iq$. We thus have:

$$B_i \geq A_j \text{ with } j = \left\lceil \frac{iq-d}{p} \right\rceil \tag{1}$$

This equation characterizes the data-dependency between A and B. However, because of the ceiling function, this equation is not suited to symbolic manipulations. We propose in Section IV a new characterization that is more intuitive and suitable to reason about buffer sizes.

III. Throughput and Duality

In this section, we first determine the exact maximal throughput for acyclic SDF graphs. Then, we present a new property that will be used to answer the minimum buffer sizes question.

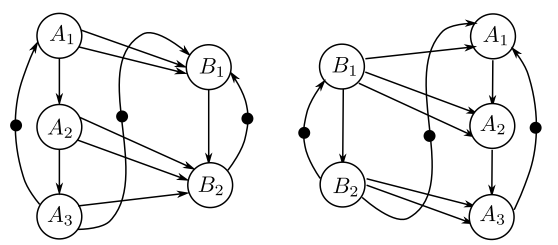
Property 1 (Throughput)**.** *The maximal throughput of an acyclic SDF graph $G = (V, E)$ is equal to*

$$\mathcal{T}_G = \frac{1}{\max\limits_{A \in V}\{z_A t_A\}} \tag{2}$$

Hence, the minimal period is $\mathcal{P}_G = \max\limits_{A \in V}\{z_A t_A\}$.

Proof: This is easily shown by considering the MCM analysis of the corresponding HSDF graph. The only cycles in *HSDF*(G), the HSDF graph equivalent to an acyclic graph G, are those used to represent the infinite firings of the same actor (see Fig. 3(a)). For each actor A, its corresponding cycle contains one delay and z_A instances of A. Thus, the cycle mean is equal to $z_A t_A$, the MCM is equal to $\max\limits_{A \in V} z_A t_A$ and denotes the inverse of the maximal throughput of *HSDF*(G) and G. ■

We say that actor A imposes a higher load than actor B when $z_A t_A > z_B t_B$. The throughput and period of an acyclic graph is defined by the actor which has the highest load. This implies that this actor never gets idle once the execution enters the steady state.



(a) HSDF graph equivalent to $A \xrightarrow[2 \quad 3]{\bullet} B$ (b) HSDF graph equivalent to $B \xrightarrow[3 \quad 2]{\bullet} A$

Fig. 3: SDF-to-HSDF transformation of a graph and its dual.

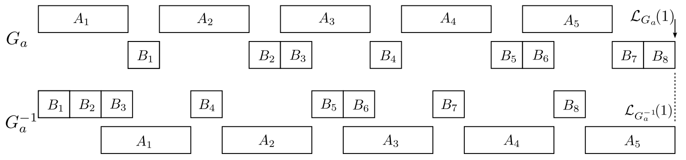


Fig. 4: Illustration of the duality theorem.

The *dual* of an SDF graph G, denoted G^{-1}, is obtained by reversing all edges of G.

Theorem 1 (Duality theorem)**.** *Let G be any (cyclic or not) live graph and G^{-1} be its dual, then $\mathcal{T}_G = \mathcal{T}_{G^{-1}}$ and $\forall i.\ \mathcal{L}_G(i) = \mathcal{L}_{G^{-1}}(i)$.*

Proof: The detailed proof can be found in [6]. We first prove that the graph *HSDF*(G) is the dual (after renaming actors) of *HSDF*(G^{-1}) (see Fig. 3). Therefore, both HSDF graphs have the same MCM and, by Eq. (2) $\mathcal{T}_G = \mathcal{T}_{G^{-1}}$. We prove that $\forall i.\ \mathcal{L}_{HSDF(G)}(i) = \mathcal{L}_{HSDF(G^{-1})}(i)$ by unfolding both HSDF graphs for i iterations. ■

Fig. 4 illustrates the duality theorem with the SDF graph G_a of Fig. 2(b). The latency of the first iteration of the ASAP execution of that graph is equal to the latency of the first iteration of its dual *i.e.,* $\mathcal{L}_{G_a}(1) = \mathcal{L}_{{G_a}^{-1}}(1)$.

We use the transformation of a graph to its dual as well as the associated theorem at several occasions during the analysis of minimal buffer sizes.

IV. The parametric graph $A \xrightarrow{p \quad q} B$

This section focuses on the simplest SDF graph made of a single edge: $G = \{A \xrightarrow{p \quad q} B\}$. It provides *exact* answers for the minimum buffer sizes needed to achieve maximal throughput. Graph G is parametrized by production and consumption rates[1] $p, q \in \mathbb{N}^+$ as well as execution times $t_A, t_B \in \mathbb{R}^+$. This section shows that the symbolic analysis, even for such simple graphs, is quite involved.

The balance equation $z_A p = z_B q$ entails that the repetition vector of this graph is:

$$[z_A = q/\gcd(p,q), z_B = p/\gcd(p,q)]$$

and, according to Property 1, its period is:

$$\mathcal{P}_G = \max(z_A t_A, z_B t_B) \tag{3}$$

A. Enabling patterns

We introduce *enabling patterns* which characterize the data-dependency between a producer and a consumer. Compared to Eq. (1), they are more intuitive and suitable to reason about buffer sizes.

Enabling patterns between the producer A and consumer B are defined by the following grammar:

$$P \quad ::= \quad A^i \rightsquigarrow B^j \quad | \quad [P]^{x=1..k} \quad | \quad P_1; P_2$$

where i, j, k evaluate to a positive integer.

An enabling pattern P is either a basic pattern ($A^i \rightsquigarrow B^j$), a repetition for k times ($[P]^{x=1..k}$), or a sequence of enabling patterns $P_1; P_2$. The expressions i, j or k are arithmetic

[1]Mathematically speaking, the results of this section can be generalized to rates defined as positive real numbers.

Case A. $p \geq q$

Let $p = kq + r$ with $0 \leq r < q$

Case A.1. $r = 0$

$$A \rightsquigarrow B^k \tag{4}$$

Case A.2. $q \leq 2r$

$$\left[A \rightsquigarrow B^k; \left[A \rightsquigarrow B^{k+1}\right]^{\alpha_j}\right]^{j=1\cdot\cdot\frac{q-r}{\gcd(p,q)}} \tag{5}$$

Case A.3. $q > 2r$

$$\left[\left[A \rightsquigarrow B^k\right]^{\beta_j}; A \rightsquigarrow B^{k+1}\right]^{j=1\cdot\cdot\frac{r}{\gcd(p,q)}} \tag{6}$$

where $\alpha_j = \left\lfloor \frac{jr}{q-r} \right\rfloor - \left\lfloor \frac{(j-1)r}{q-r} \right\rfloor$ and $\beta_j = \left\lceil \frac{jq}{r} \right\rceil - \left\lceil \frac{(j-1)q}{r} \right\rceil - 1$.

Case B. $p < q$

Let $q = kp + r$ with $0 \leq r < p$

Case B.1. $r = 0$

$$A^k \rightsquigarrow B \tag{7}$$

Case B.2. $p \geq 2r$

$$\left[A^{k+1} \rightsquigarrow B; \left[A^k \rightsquigarrow B\right]^{\gamma_j}\right]^{j=1\cdot\cdot\frac{r}{\gcd(p,q)}} \tag{8}$$

Case B.3. $p < 2r$

$$\left[\left[A^{k+1} \rightsquigarrow B\right]^{\lambda_j}; A^k \rightsquigarrow B\right]^{j=1\cdot\cdot\frac{p-r}{\gcd(p,q)}} \tag{9}$$

where $\gamma_j = \left\lfloor \frac{jp}{r} \right\rfloor - \left\lfloor \frac{(j-1)p}{r} \right\rfloor - 1$ and $\lambda_j = \left\lceil \frac{jr}{p-r} \right\rceil - \left\lceil \frac{(j-1)r}{p-r} \right\rceil$.

Fig. 5: Enabling patterns.

expressions made of integers, parameters or pattern variables defined by enclosing repetition patterns.

The semantics of an enabling pattern is defined *w.r.t.* two counters c_A and c_B representing the number of completed firings of A and B (initially 0). The pattern $A^i \rightsquigarrow B^j; P$ *w.r.t.* (c_A, c_B) means that:

- c_A firings of A have produced enough tokens to fire actor B exactly c_B times;
- then, if A is not fired at least i times more then B cannot be fired; otherwise B can be fired j times;
- the subsequent pattern P is considered with the new values $(c_A + i, c_B + j)$.

A repetition $[P]^{x=1..k}$ is a sequence of k patterns P. The pattern $[P]^{x=1..k}$ is also written $[P]^k$ if the pattern variable x is not used in P.

A correct enabling pattern must cover an entire iteration: at the end of the pattern, we should have $c_A = z_A$ and $c_B = z_B$. For instance, the enabling pattern of $A \xrightarrow{3 \quad 6} B$ is $A^2 \rightsquigarrow B$; *i.e.,* after every two firings of actor A, one firing of B is enabled. The enabling pattern of $A \xrightarrow{8 \quad 5} B$ is:

$$A \rightsquigarrow B; A \rightsquigarrow B^2; A \rightsquigarrow B; [A \rightsquigarrow B^2]^2$$

which is illustrated in Fig. 6. It can also be written as:

$$\left[A \rightsquigarrow B; [A \rightsquigarrow B^2]^i\right]^{i=1\cdot\cdot 2}$$

This representation is very useful when the length and shape of enabling patterns depend on the parameters of the graph.

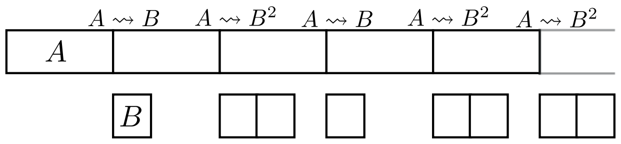


Fig. 6: An ASAP execution ($p = 8$, $q = 5$, $t_A = 20$, $t_B = 7$).

B. Enabling patterns of $A \xrightarrow{p \quad q} B$

Depending on the production and consumption rates p and q, there are six possible enabling patterns.

Property 2. *Fig. 5 gathers all possible enabling patterns of a channel $A \xrightarrow{p \quad q} B$.*

Proof: We show here the proof of cases (A.1) and (A.2). The remaining cases can be found in [6].

• *Case (A.1)* If $r = 0$ then $p = kq$ and the repetition vector is $\vec{z} = [1, k]$. Each firing of A enables k firings of B and the enabling pattern is $A \rightsquigarrow B^k$.

• *Case (A.2)*($p = kq + r$ and $q \leq 2r$) The first firing of A enables only k firings of B, yielding the pattern $A \rightsquigarrow B^k$. The number of remaining tokens in the channel after sequence AB^k is equal to r. Since $2r \geq q$, the second firing of A enables $k + 1$ firings of B, yielding the aggregated pattern $A \rightsquigarrow B^k; A \rightsquigarrow B^{k+1}$. It follows that pattern $A \rightsquigarrow B^{k+1}$ can be repeated a number of times denoted α_1, after which there will be $r + \alpha_1(r - q)$ tokens left. So, α_1 is the largest integer such that $r + \alpha_1(r - q) \geq 0$. Hence, $\alpha_1 = \lfloor \frac{r}{q-r} \rfloor$.

The next firing of A will only enable k firings of B ($A \rightsquigarrow B^k$). By the same reasoning as above, this will be followed by $[A \rightsquigarrow B^{k+1}]^{\alpha_2}$ where α_2 is the largest integer such that $r + \alpha_1(r-q) + r + \alpha_2(r-q) \geq 0$. Hence, $\alpha_2 = \lfloor \frac{2r}{q-r} \rfloor - \lfloor \frac{r}{q-r} \rfloor$. This process is repeated until all firings of A and B of the iteration take place, yielding the complete pattern

$$\left[A \rightsquigarrow B^k; [A \rightsquigarrow B^{k+1}]^{\alpha_j}\right]^{j=1\cdot\cdot m}$$

for some m that remains to compute. The number of firings of A in this pattern is equal to $c_A = \sum_{j=1}^{m} 1 + \alpha_j$, while the number of firings of B is equal to $c_B = \sum_{j=1}^{m} k + (k+1)\alpha_j$. A correct pattern should cover the entire iteration, *i.e.,* $c_A = z_A$ and $c_B = z_B$, which implies (details omitted, see [6]) $m = \frac{q-r}{\gcd(p,q)}$. ■

The enabling patterns of Fig.5 can be used to derive a quasi-static schedule for the parametric graph $A \xrightarrow{p \quad q} B$.

C. Minimum buffer size of $A \xrightarrow{p \quad q} B$

We now use enabling patterns to compute the minimum size of the buffer $A \xrightarrow{p \quad q} B$ (denoted $\theta_{A,B}$) such that the ASAP execution achieves the maximal throughput or, equivalently, the minimal period given by Eq. (3). The buffer size is modeled by adding a backward edge with $\theta_{A,B}$ initial tokens. We distinguish two cases:

• **Case** $z_A t_A \geq z_B t_B$ (*i.e.,* $q t_A \geq p t_B$): Actor A has the highest load and should fire consecutively for maximal throughput. Enabling patterns are the key to find the minimum number of tokens that allows this behavior. A trivial case is (A.1) where $p = kq$ and the enabling pattern is $A \rightsquigarrow B^k$. In order to perform the first two firings of A consecutively, the backward edge should have at least $2p$ tokens. Since $q t_A \geq k q t_B$ (hence $t_A \geq k t_B$), the k firings of B complete before the third firing of A which still needs $2p$ initial tokens in order to fire again immediately. Hence, the minimum buffer size is $2p$.

• **Case** $z_A t_A < z_B t_B$ (*i.e.,* $qt_A < pt_B$): Actor B has the highest load and should fire consecutively for maximal throughput. However, in general all firings of B cannot be consecutive since initially, there is no token on the edge. The previous approach can still be followed using the duality theorem. Since the graph G and its dual G^{-1} have the same throughput, we can apply the former reasoning on G^{-1} where B is the producer and has the highest load.

Property 3. *If $z_A t_A \geq z_B t_B$, the minimum buffer sizes of $A \xrightarrow{p \quad q} B$ for maximal throughput are given by the symbolic formulas of Fig. 7.*

Case I.

Case I.1. A.1 $\vee$ ((A.2 $\vee$ A.3) $\wedge\, t_A \geq (k+1)t_B$)

$$\theta_{A,B} = 2p + q - \gcd(p,q) \tag{10}$$

Case I.2. B.1 $\vee$ ((B.2 $\vee$ B.3) $\wedge\, t_B \leq kt_A$)

$$\theta_{A,B} = p + q - \gcd(p,q) + \left\lceil \frac{t_B}{t_A} \right\rceil p \tag{11}$$

Case II.

Case II.1. $(\text{A.2} \wedge r' \geq \frac{\lceil \frac{r}{q-r} \rceil}{\lceil \frac{r}{q-r} \rceil + 1} t_B) \vee (\text{A.3} \wedge r' \geq \frac{1}{\lfloor \frac{q}{r} \rfloor} t_B)$ where $r' = t_A - kt_B$

$$\theta_{A,B} = 2p + q - \gcd(p,q) + \left\lceil \frac{t_B - r'}{r'} \right\rceil r \tag{12}$$

Case II.2. $(\text{B.2} \wedge r' \leq \frac{1}{\lceil \frac{p}{r} \rceil} t_A) \vee (\text{B.3} \wedge r' \leq \frac{\lfloor \frac{r}{p-r} \rfloor}{\lfloor \frac{r}{p-r} \rfloor + 1} t_A)$ where $r' = t_B - kt_A$

$$\theta_{A,B} = p + 2q - \gcd(p,q) + \left\lceil \frac{r'}{t_A - r'} \right\rceil (p - r) \tag{13}$$

Case III.

Case III.1. A.2

$$\theta_{A,B} = 2p + q + r - \gcd(p,q) + \max_{j=1}^{n-1} (jr \bmod (q-r)) \tag{14}$$

where n is the smallest positive integer such that $\left\lfloor \frac{nr'}{t_B - r'} \right\rfloor \geq \left\lceil \frac{nr}{q-r} \right\rceil$ and $r' = t_A - kt_B$.

Cases III.(A.3), III.(B.2), III.(B.3) see [6].

Fig. 7: Minimum buffer size $\theta_{A,B}$ when $z_A t_A \geq z_B t_B$.

Proof: Here, we only outline the proof and focus on one case ((I.1) with (A.2)). A detailed proof can be found in [6].

Let δ_j be the minimum number of tokens in the backward edge (representing the buffer) such that the j^{th} firing of A can occur immediately after the $(j-1)^{th}$ firing of A. By definition of $\theta_{A,B}$, we have $\theta_{A,B} = \max_j \delta_j$. Let x_j denote the number of firings of B that have finished by the start of the j^{th} firing of A. Hence, $\delta_j = jp - x_j q$.

The three cases of Fig. 7 should be read as (I) else (II) otherwise (III).

Case (I): *At any given enabling point* (i.e., *any* $\rightsquigarrow$ *in the enabling pattern), all newly enabled firings of B complete their execution before the next enabling point.*

In terms of the enabling pattern cases identified in Fig. 5, case (I) must be split into two exclusive subcases, (I.1) when $p \geq q$ and (I.2) otherwise. The conjunction of the condition for case (I) with $p \geq q$ yields the condition A.1 $\vee$ ((A.2 $\vee$ A.3) $\wedge\, t_A \geq (k+1)t_B$), which is shown in Fig. 7. The other conditions are obtained similarly.

Let us prove the result for case (I.1) and pattern (A.2), *i.e.,* $p = kq + r \wedge q \leq 2r$. According to the enabling pattern (Eq. (5)), at most $(k+1)$ firings of B can be enabled at a given point. They all run in parallel with one firing of A. Case (I) requires that $t_A \geq (k+1)t_B$.

We first expand the enabling pattern of Eq. (5) into an infinite pattern in order to compute the minimum buffer size $\theta_{A,B}$ over the infinite execution of the graph:

$$\left[A \rightsquigarrow B^k;\, \left[A \rightsquigarrow B^{k+1}\right]^{\alpha_j}\right]^{j \in \mathbb{N}^+} \tag{15}$$

For the sake of the proof, since $\alpha_j \geq 1$, we can rewrite Eq. (15) into the equivalent infinite pattern:

$$\underbrace{A \rightsquigarrow B^k; A \rightsquigarrow B^{k+1}}_{\text{prologue}};\underbrace{\left[\left[A \rightsquigarrow B^{k+1}\right]^{\alpha_j - 1}; A \rightsquigarrow B^k; A \rightsquigarrow B^{k+1}\right]}_{\text{block } j}{}^{j \in \mathbb{N}^+} \tag{16}$$

The minimum of tokens to enable the first firing of A must be $\delta_1 = p$. Then, $\delta_2 = \delta_1 + p = 2p$ because no B has finished before the second A starts. Then, $\delta_3 = \delta_2 + (p - kq) = 2p + r$. Subsequently, $\delta_4 = \delta_3 + (p - (k+1)q) = \delta_3 + (r - q)$, and all α_1 subsequent values of δ_i are obtained from δ_{i-1} by adding $(r - q)$ which is negative. The sequence (α_j) is defined in Fig. 5. Hence, $\delta_{1+(\alpha_1+1)+1} = \delta_3 + \alpha_1(r-q) = 2p + r + \alpha_1(r-q)$. This ends at the last of the α_1 patterns $A \rightsquigarrow B^{k+1}$, so the next value $\delta_{1+(\alpha_1+1)+2}$ is obtained by adding $p - kq = r$ because of pattern $A \rightsquigarrow B^k$, yielding $2p + r + \alpha_1(r - q) + r$.

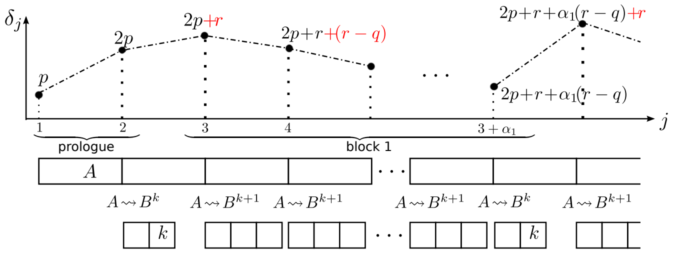


Fig. 8: Sequence (δ_j) in case (I.1) and pattern (A.2).

The computation of the infinite sequence (δ_j) is illustrated in Fig. 8. Within each block j, the subsequence (δ_h) is strictly decreasing because $(r - q)$ is negative, so its maximum value is the value of the entry point, which we denote by $\ell_j = 1 + \sum_{i=1}^{j}(\alpha_i + 1)\; + 2$. We thus have:

$$\delta_{\ell_j} = 2p + r + \sum_{i=1}^{j} (\alpha_i(r-q) + r) = 2p + r + jr + (r-q) \left\lfloor \frac{jr}{q-r} \right\rfloor$$

It follows that the maximum value of the infinite sequence (δ_j) is:

$$\begin{aligned}\theta_{A,B} &= \max_{j\in\mathbb{N}} \delta_{\ell_j} \\ &= \max_{j\in\mathbb{N}}(2p + r + jr + (r-q)\lfloor \frac{jr}{q-r} \rfloor) \\ &= 2p + r + (q-r)\max_{j\in\mathbb{N}}(\frac{jr}{q-r} - \lfloor \frac{jr}{q-r} \rfloor)\end{aligned}$$

As a conclusion, $\theta_{A,B} = 2p + r + (q - r - \gcd(p, q)) = 2p + q - \gcd(p, q)$.

CASE (II): *Case (I) is not satisfied, but, for any block (*e.g., $[[A \rightsquigarrow B^{k+1}]^{\alpha_j}; A \rightsquigarrow B^k]$ *in case (A.2)), all firings of B during this block complete their execution before the first enabling point in the next block.*

This case is illustrated in Fig. 9. A block is of the form $[A \rightsquigarrow B^2]^{\alpha_j}; A \rightsquigarrow B$. The maximum value of α_j is 2. Therefore, five firings of B have to run in parallel with three firings of A. So, we must have $5t_B \leq 3t_A$. The computed sequence (δ_j) in case (II) is similar to that of case (I) but with small shifts.

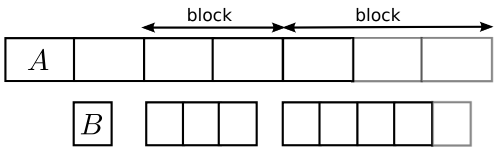


Fig. 9: An ASAP execution (p=8, q=5, t_A=13, t_B=7).

CASE (III): *Otherwise.*

This is the most complicated case to solve since the sequence (x_j) does not follow the enabling patterns. Our solution is based on the following observations. We define a catch-up sequence as a sequence of consecutive firings of B (*i.e.,* without gaps) that may spread over many blocks. Fig. 10 illustrates a catch-up sequence over two blocks.

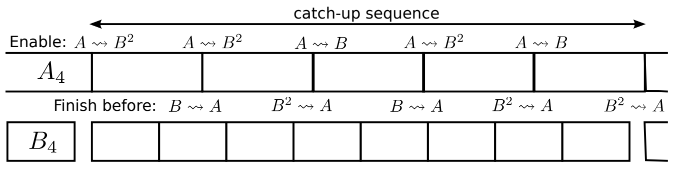


Fig. 10: An ASAP execution (p=8, q=5, t_A=23, t_B=14).

The key observation is the following. For the firings of A inside a catch-up sequence, the number of firings of B that finish before firings of A actually follows the enabling pattern of graph $A \xrightarrow{t_A \quad t_B} B$, *i.e.,* as if *time* was produced and consumed instead of tokens. Furthermore, the maximum of sequence (δ_j) occurs inside the maximal (in terms of blocks) catch-up sequence. For instance, n in Eq. (14) represents the length of the maximal catch-up sequence. ■

In Case (III), when actors A and B impose the same load (*i.e.,* $z_A t_A = z_B t_B$), the catch-up sequence spreads all over the iteration. In this worst-case scenario, all four cases (III.A.2, III.A.3, III.B.2 and III.B.3) give the same *upper bound*:

$$\theta^u_{A,B} = 2(p + q - \gcd(p, q)) \tag{17}$$

This bound is also tight, in the sense that for all p, q, there exist t_A and t_B such that $\theta_{A,B}$ given in Eq. (14) is equal to $\theta^u_{A,B}$. This upper bound does not depend on the execution times of the actors. Therefore, it can be used as a safe buffer size if the execution times of actors are unknown.

Our last result concerns the minimum buffer size of the graph $A \xrightarrow{p \quad q} B$ when there are d initial tokens.

Property 4. *If If $z_A t_A \geq z_B t_B$ and the channel $A \xrightarrow{p \quad q} B$ contains d initial tokens, then the minimum buffer size $\theta'_{A,B}$ that allows the maximum throughput is*

$$\theta'_{A,B} = \max\{0, \theta_{A,B} - d + d \bmod \gcd(p, q)\} \tag{18}$$

with $\theta_{A,B}$ as defined in Fig. 7

Proof: See [6]. ■

If $z_A t_A < z_B t_B$ the minimum buffer sizes are computed identically but on the dual graph.

V. ACYCLIC GRAPHS

Exact symbolic analyses for a single edge are already so complex that they seem to be out of reach for arbitrary (even acyclic) graphs. This section shows how to use the previous results to obtain *approximate* analyses for the minimum buffer sizes of general acyclic dataflow graphs. To achieve this, we make use of a technique that linearizes the firings of actors.

A. Linearization of graph $A \xrightarrow{p \quad q} B$

Consider the graph $G = \{A \xrightarrow{p \quad q} B\}$, where, as illustrated in Fig. 6, the firings of A are consecutive while those of B are neither consecutive nor uniformly distributed. Let $f_B(i)$ denote the finish time of the i^{th} firing of actor B. In order to derive formulas that can be composed (*e.g.,* when dealing with a chain of actors), we want to transform B into a fictive actor B^u that fires consecutively as many times as B and such that

$$\forall i.\ f_B(i) \leq f_{B^u}(i)$$

Actor B^u has a starting time $t^0_{B^u}$ and an execution time t_{B^u}, and since it fires consecutively $f_{B^u}(i) = i t_{B^u} + t^0_{B^u}$.

We present one linearization method, called *Stretch*, illustrated in Fig. 11. Method *Stretch* determines B^u by increasing the execution time of B in order to fill the gaps. We distinguish two cases:

- **Case** $qt_A \geq pt_B$: We take $t_{B^u} = \frac{qt_A}{p}$. The starting time can be shown (see the proof in [6]) to be $t^0_{B^u} = t_A + t_B - \frac{\gcd(p,q)}{p} t_A$. So,

$$\forall i.\ f_{B^u}(i) = \frac{qt_A}{p} i + \left(t_A + t_B - \frac{\gcd(p,q)}{p} t_A\right) \tag{19}$$

Method *Stretch* may advance the starting of some firings, but always postpone their endings.

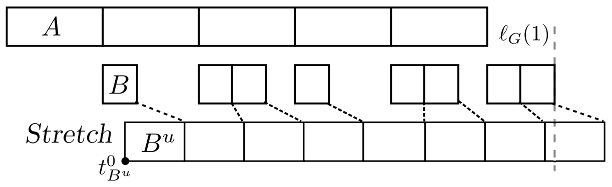


Fig. 11: Linearization of B for the graph $A \xrightarrow{p \quad q} B$ (p = 8, q = 5, t_A = 20, t_B = 7). Case $qt_A \geq pt_B$.

• **Case** $qt_A < pt_B$: Firings of B are consecutive in the steady state. Therefore, we can take $t_{B^u} = t_B$ and using the duality theorem, we have $\mathcal{L}_G(n) = \mathcal{L}_{G^{-1}}(n) = nz_Bt_B + \Delta_{B,A}$ and we can take $t^0_{B^u} = \Delta_{B,A}$, where $\Delta_{B,A}$ is computed on the schedule of the first iteration of the dual graph $B \xrightarrow{q \quad p} A$ as the difference between the end of the last firing of A and the end of the last firing of B (see [6] for details).

In both cases, it can be shown that this linearization is tight in the sense that $\exists i.\ f_B(i) = f_{B^u}(i)$. We can see in Fig. 11 that both 5th firings of B and B^u finish at the same time.

Thanks to this linearization, the chain $A \xrightarrow{p \quad q} B \xrightarrow{p' \quad q'} C$ can be treated by first scheduling the subgraph $A \xrightarrow{p \quad q} B$, then linearizing the firings of B if they are not consecutive, then scheduling the subgraph $B^u \xrightarrow{p' \quad q'} C$, and finally combining the two schedules.

B. Minimum buffer sizes for maximal throughput

We give approximate minimum buffer sizes that allow general acyclic graphs to reach their maximal throughput. We first present formulas to compute safe upper bounds for general acyclic graphs, then we present a heuristic that improves this bound for chains, trees, and in-trees (DAGs with only joins). These special kinds of graphs, especially chains, are common in streaming applications.

1) Safe upper bounds:
We first present a negative result. Let G be an acyclic graph and let the size of each channel $A \xrightarrow{p \quad q} B$ be equal to $\theta_{A,B}$ as defined in Section IV-C. These buffer sizes do not always permit maximal throughput[2]. A simple counterexample is the graph $G_b = \{A \xrightarrow{3 \quad 4} B \xrightarrow{4 \quad 2} C\}$ with $t_A = 16$, $t_B = 11$ and $t_C = 12$. The repetition vector is $\vec{z} = [4, 3, 6]$. Actor C imposes the higher load, hence the minimal period of this graph is $\mathcal{P}_{G_b} = z_Ct_C = 72$. We have $\theta_{A,B} = 9$ and $\theta_{B,C} = 6$. Locally, these buffer sizes allow the producers to run freely without any constraint from the consumers. However, when they are put together, the maximal throughput, where actor C fires consecutively, cannot be achieved (see Fig. 12). The computation of $\theta_{B,C}$ assumes that the execution time of actor B is $t_B = 11$. However, as illustrated in Fig. 12, there are gaps between the firings of B due to the data-dependency $A \to B$. The global execution proceeds as if the execution times of B were sometimes longer than 11.

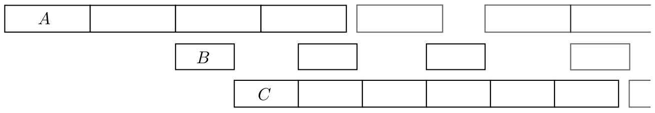


Fig. 12: ASAP execution of G_b

Property 5. *Let G be a graph without any undirected cycle, if the buffer of every channel $A \xrightarrow{p \quad q} B$ in G is at least $2(p + q - \gcd(p, q))$, then the ASAP execution of the graph achieves the maximal throughput.*

Proof: We first present the proof for chains. Let G be the chain $\{A_1 \xrightarrow{p_1 \quad q_1} A_2 \xrightarrow{p_2 \quad q_2} A_3 \to \cdots \to A_n\}$, according to Eq. (2), the minimal period of G is $\mathcal{P}_G = \max\limits_{i=1..n}\{z_{A_i}t_{A_i}\}$.

[2]They do however allow maximal throughput in some specific cases described in the next section

The period and therefore the throughput remain the same if the execution time of each actor A_i is considered to be $\frac{\mathcal{P}_G}{z_{A_i}}$. Let $G_=$ be the version of G where all actors have the same load as the maximum load in G. Then G and $G_=$ have the same period and throughput.

If the size of each buffer $A_i \xrightarrow{p_i \quad q_i} A_{i+1}$ in $G_=$ is $\theta^u_{A_i,A_{i+1}} = 2(p_i + q_i - \gcd(p_i, q_i))$, then $G_=$ still achieves the maximal throughput. Indeed, size $2(p_1 + q_1 - \gcd(p_1, q_1))$ for the first channel allows both A_1 and A_2 to run consecutively in the steady state (see Eq. (17)). Similarly, size $2(p_2 + q_2 - \gcd(p_2, q_2))$ for the second channel allows both A_2 and A_3 to run consecutively, and so on.

Since graph $G_=$ with these buffer sizes achieves the maximal throughput, reducing the execution times of actors in $G_=$ to their original values will never decrease the throughput of the graph thanks to the monotonicity of the self-timed execution. Hence, graph G with these buffer sizes achieves the maximal throughput.

The proof for general graphs with no undirected cycle can be found in [6]. ■

Note 1: Since the minimum buffer sizes below which the graph is definitely not live are equal to $p+q-\gcd(p, q)$ [1], Property 5 provides a first solution which is less than twice the exact one. For parametric dataflow models, the upper bound $\theta^u_{A,B}$ can actually be reached for some configurations. However, if the system supports dynamic reallocation of memory, it is still useful to evaluate the minimal buffer sizes to adjust buffers after each configuration change.

Unfortunately, Property 5 does not hold for general acyclic graphs that contain undirected cycles. A counterexample is the graph $G_c = \{A \xrightarrow{4 \quad 3} B \xrightarrow{3 \quad 8} D,\ A \xrightarrow{1 \quad 3} C \xrightarrow{3 \quad 2} D\}$ with $t_A = 4$, $t_B = 3$, $t_C = 12$ and $t_D = 8$. The repetition vector is $\vec{z} = [6, 8, 2, 3]$ and all actors impose the same load (*i.e.,* $\forall X.\ z_Xt_X = 24$). The ASAP execution when all buffer sizes are equal to their upper bound $2(p + q - \gcd(p, q))$ is shown in Fig. 13. Actor A does not fire consecutively so the throughput is not maximal. The reason is that the chain $A \to C \to D$ imposes an earliest start time for D that is after the earliest start time imposed by the chain $A \to B \to D$. More precisely, the first firing of actor D is delayed by actor C (*i.e.,* by the second chain), which delays the 7^{th} firing of B which in turn delays the 8^{th} firing of A. Let $f^u_{D,1}$ (resp. $f^u_{D,2}$) denote the linear upper bound on finish times of actor D following the first (resp. second) chain. We have $f^u_{D,1}(i) = 8i + 16$ and $f^u_{D,2}(i) = 8i + 28$, hence $f^u_{D,2}(i) > f^u_{D,1}(i)$. In order to prevent the second chain from impacting the schedule of the first chain, we must increase the size of buffer $B \to D$ so that B may fire during $28 - 16$ time units. Since B produces 3 tokens and $t_B = 3$ the size of the $B \to D$ buffer must be increased by $\lceil\frac{28-16}{3}\rceil 3 = 12$.

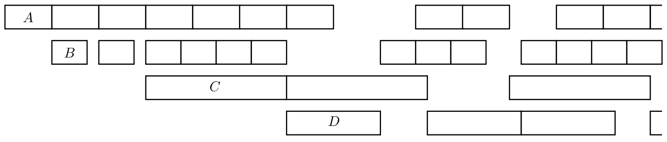


Fig. 13: ASAP execution of G_c

In the general case, for a chain $A_1 \xrightarrow{p_1 \quad q_1} A_2 \to \cdots \xrightarrow{p_{n-1} \quad q_{n-1}} A_n$ where all actors have the same load, we have

$$f^u_{A_n}(i) = t_{A_n} i + z_{A_n} t_{A_n} \sum_{i=1}^{n-1} \frac{p_i + q_i - \gcd(p_i, q_i)}{p_i z_{A_i}} \quad (20)$$

If the actors do not have the same load, we consider, as in the proof of Property 5, the chain where all actors have the same load *i.e.,* the maximum load of the original chain.

Property 6. *Let two different chains from A_1 to A_n such that $f^u_{A_n,1}(i) = t_{A_n} i + x_1$, $f^u_{A_n,2}(i) = t_{A_n} i + x_2$ and $x_1 < x_2$. To prevent the second chain from disturbing the schedule of the first one, it suffices to increase the size of the last channel $A_{n-1} \xrightarrow{p \quad q} A_n$ of the first chain by ζ with*

$$\zeta = \left\lceil \frac{x_2 - x_1}{t_{A_{n-1}}} \right\rceil p \quad (21)$$

Proof: The proof is based on the following observation. Let $A \xrightarrow{p \quad q} B$ be a graph with two actors such that $z_A t_A = z_B t_B$. Hence, $\theta_{A,B} = 2(p + q - \gcd(p, q))$. Even if the first firing of B starts only at time $\frac{t_A}{p}(p + q - \gcd(p, q))$, actor A can still fire consecutively provided that the buffer size is $\theta^u_{A,B}$ (the proof uses Eq. (19), details omitted). Hence, if all actors in the first chain are delayed as indicated by Eq. (20), then the sizes $2(p_i + q_i - \gcd(p_i, q_i))$ still allow the first actor to fire consecutively. Note that after introducing these delays, no actor gets idle once it starts executing.

Let x_1 and x_2 denote the start time of the last actor in the two different chains from A_1 to A_n. If $x_2 > x_1$, the extra delay $(x_2 - x_1)$ imposed by the second chain will hinder the previous property. Let $A_{n-1} \xrightarrow{p \quad q} A_n$ be the last channel in the first chain. Increasing the size of this channel by $\lceil \frac{x_2 - x_1}{t_{A_{n-1}}} \rceil p$ will allow actor A_{n-1} to fire consecutively during the extra delay $(x_2 - x_1)$. That is, the impact of the second chain on the first one is avoided. ■

Note 2: The value of ζ, like the value of $\theta^u_{A,B}$, does not actually depend on execution times. Indeed, Eq. (20) shows that x_1 and x_2 can be expressed as $z_{A_n} t_{A_n} k_1$ and $z_{A_n} t_{A_n} k_2$. Since $z_{A_n} t_{A_n} = z_{A_{n-1}} t_{A_{n-1}}$, term $\frac{x_2 - x_1}{t_{A_{n-1}}}$ can be expressed as $z_{A_{n-1}}(k_2 - k_1)$ which does not depend on execution times.

This approach can be extended to deal with any acyclic graph. The final property should be that for any two different chains from A_1 to A_n in the graph such that

$$f^u_{A_n,1}(i) = t_{A_n} i + x_1, \quad f^u_{A_n,2}(i) = t_{A_n} i + x_2 \text{ and } x_1 < x_2$$

the size of the last channel $A_{n-1} \xrightarrow{p \quad q} A_n$ of the "fastest" chain is *at least* equal to $2(p + q - \gcd(p, q)) + \lceil \frac{x_2 - x_1}{t_{A_{n-1}}} \rceil p$. This property can be enforced by computing, for each node X and each chain from a source actor to X, the function $f^u_{X,j}(i) = t_X i + x_j$. Then, as in Property 6, we add to each final edge of these chains the value ζ computed *w.r.t.* the maximum computed x_j. Such an approach is safe but not always needed (*e.g.,* when the predecessors of a node do not have common ancestors). We do not describe the refined and optimized algorithm here.

2) Improving the upper bounds:

In this section, we improve the minimum buffer sizes for chains, trees, and in-trees, starting with chains. We say that a chain is *monotone* if each actor imposes a higher load than its successor or if each actor imposes a lower load than its successor.

Definition 1. *The chain $A_1 \to \cdots \to A_n$ is* monotone *if and only if $(\forall i.\ z_{A_i} t_{A_i} \geq z_{A_{i+1}} t_{A_{i+1}}) \vee (\forall i.\ z_{A_i} t_{A_i} \leq z_{A_{i+1}} t_{A_{i+1}})$*

Let $\theta_{A_i, A_{i+1}}$ be the size of the buffer between A_i and A_{i+1} as computed in Section IV-C. This size allows the single edge graph $A_i \xrightarrow{p_i \quad q_i} A_{i+1}$ to reach its maximal throughput.

Property 7. *A monotone chain $A_1 \to \cdots \to A_n$ where the size of each buffer $A_i \to A_{i+1}$ is at least $\theta_{A_i, A_{i+1}}$ achieves its maximal throughput.*

Proof: Suppose that the chain has the descending order $\forall i.\ z_{A_i} t_{A_i} \geq z_{A_{i+1}} t_{A_{i+1}}$. The size θ_{A_1, A_2} allows actor A_1 to run consecutively, which is the same behavior as when there is no constraint on buffer sizes. Then, the size θ_{A_2, A_3} allows actor A_2 to fire as soon as it is enabled by actor A_1. Actually, if there were no dependence from A_1 to A_2, the size θ_{A_2, A_3} would allow actor A_2 to run consecutively. The same reasoning shows that all actors are fired as it there were no buffer size constraints. Therefore, the chain achieves its maximal throughput. The case of ascending order can be easily dealt with using the duality theorem. ■

Note that Property 7 is only a *sufficient* condition simply because $\theta_{A_i, A_{i+1}}$ allows actor A_i to fire consecutively. However, as soon as when $i > 1$, it is not always necessary in order to achieve the maximal throughput. E.g., let $G_d = \{A_1 \xrightarrow{2 \quad 2} A_2 \xrightarrow{4 \quad 3} A_3\}$ such that $t_{A_1} = 28$, $t_{A_2} = 20$ and $t_{A_3} = 15$. Though $\theta_{A_2, A_3} = 12$, the minimum size of channel $A_2 \to A_3$ is actually 9. Indeed, as illustrated in Fig. 14, even if this size delays firings of A_2 (see the 4th firing), the introduced delay does not prohibit A_1 from firing consecutively.

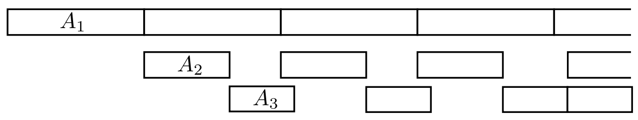


Fig. 14: ASAP execution of G_d.

Property 7 also holds for non monotone chains made of an ascending sub-chain followed by a descending one. We say that those chains are of the form $\sqcap$. The computed buffer sizes on both wings allow the actors at the top to run consecutively.

Unfortunately, as illustrated in Fig. 12 in the previous section, Property 7 does not hold for any chain. Our solution is hence to put any chain on the form $\sqcap$ by using the same approach as proof of Property 7; *i.e.,* by increasing the execution times of some actors (without exceeding the maximum load $\mathcal{P}_G$), then computing the buffer sizes as in Property 7, and finally restoring the original execution times. Fig. 15 illustrates this solution.

Any chain on the form $\sqcap$ obtained by increasing the load of the actors of the original chain is a valid solution. For example, the chain $A \to B \to \ldots \to K$ can be transformed in the red chain which is of the form $\sqcap$. Actually, any chain of this form inside the gray area is a valid solution. However, we can choose one that minimizes the buffer sizes. According to Section IV-C, the channel $I \to J$ is in case (I.1) (Fig. 7) and the size of the channel is 4 whatever the chosen load. The channel $J \to K$ is in case (III) and increasing the load of J may put it in case (I.1) where the buffer size is smaller. Increasing loads serve to

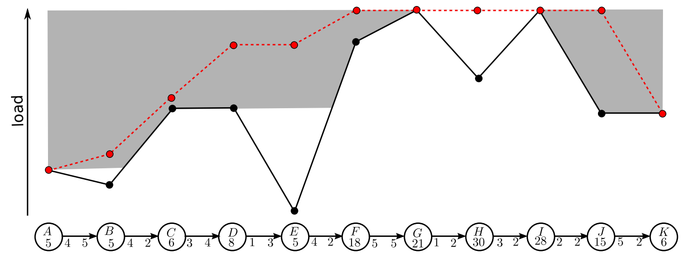


Fig. 15: Transformation of a chain to a $\sqcap$ form.

put the channel in a less expensive case as described in Fig. 7. Furthermore, as long as the load remains in the gray area, several choices can be made *e.g.,* to facilitate the treatment of subsequent channels. There are many heuristics using this idea to minimize buffer sizes. They involve other criteria such as the expected size gain that can be used to prioritize the treatment of channels. We do not describe them here but one of them is evaluated in the experiments section (see Section VI).

The case of trees is solved in the same way. If the tree does not contain any sub-trees (*i.e.,* it consists of a set of chains originating from the same root node), then the load of the root node is first increased to be equal the maximum of all loads in the tree and then the previous method can be applied on every chain composing the tree. This is correct because the computed buffer sizes will allow the root actor to run consecutively. If the tree contains sub-trees, the same process is first applied recursively on sub-trees and then it proceeds with each sub-tree replaced by its root node.

VI. Experiments

In this section, we compare the results of our heuristic presented in Section V-B2, with the safe upper bounds (Section V-B1) and the exact minimum buffer sizes using many randomly generated SDF graphs and some real benchmarks.

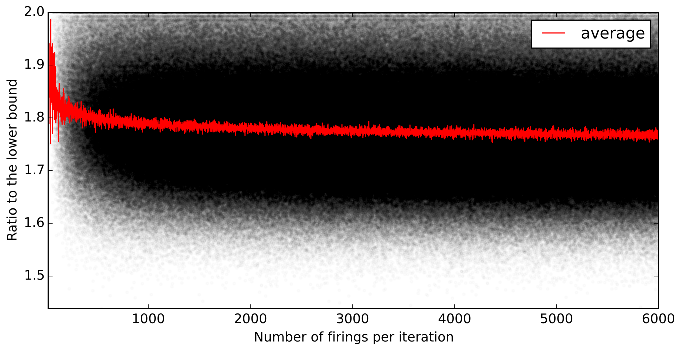


Fig. 16: Heuristic vs. upper bounds.

First, we compare the results of the heuristic with the safe upper bounds $2(p+q-\gcd(p,q))$ using two million randomly generated chains of 10 actors where production and consumption rates (resp. execution times) are uniformly distributed over the interval $[1,20]$ (resp. $[1,200]$). The number of firings per iteration, $\sum_X z_X$, of every generated chain has been bounded by 6×10^3. For each graph, we compute the ratio of the total buffers sizes obtained by our approach to the sum of the lower bounds, $p+q-\gcd(p,q)$[3]. Each black dot in Fig.16 represents the obtained ratio for one graph, while the red line represents the average of ratios. Fig. 16 shows that, in average, our heuristic reduces the total buffer sizes by 20% compared to the upper bounds.

Secondly, we compare the results of the heuristic with the exact minimum buffer sizes. The exact solution is obtained using a dichotomic search that first checks (using symbolic simulation) whether there is a channel capacities distribution, whose total size is at the middle between the lower bound and the already obtained solution by the heuristic, which allows the graph to achieve its maximal throughput. Depending on the answer of this request, the algorithm proceeds to either the top or bottom part of the search space, and so on. Due to the exponential complexity of the minimum buffer sizes problem, we evaluate our approach on only 10^4 randomly generated chains of four actors where production and consumption rates (resp. execution times) are uniformly distributed over the interval $[1,10]$ (resp. $[1,100]$). The blue (resp. red) line in Fig. 17 represents the average of the ratios of the exact (resp. approximate) solution to the lower bound. So, in average, our heuristic over-approximates the exact solution by 25%. Furthermore, Fig. 17 also shows that the exact solution is 30% above the lower bound. Hence, by extrapolating this result, our heuristic would over-approximate the exact solutions of Fig. 16 by 35%.

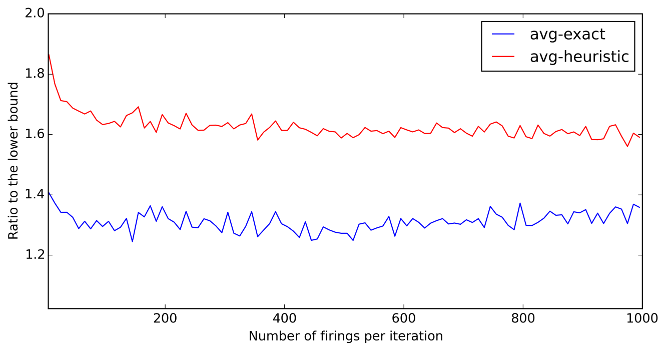
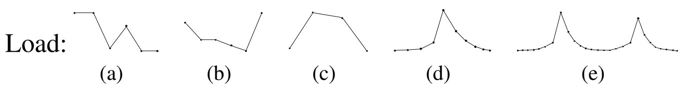


Fig. 17: Heuristic vs. exact solution.

Finally, we evaluate the heuristic using five real applications: the H.263 decoder, the data modem and sample rate converter from the SDF[3] benchmarks [17], the fast fourier transformer (FFT), and the time delay equalizer (TDE) from the StreamIt benchmarks [18]. All these graphs have a chain structure. Table I shows some characteristics of these applications together with the obtained results. Our approach improves better the upper bounds in case of chains with a $\sqcap$ form (H.263 decoder and FFT). It comes close to the upper bound for the sample rate converter since the two actors with the highest loads are the right and left ends of the chain; increasing the loads of the other actors to get a monotone chain results in a size of almost $2(p+q-\gcd(p,q))$ for every channel.

TABLE I: Experimental results for real benchmarks.

graph	# actors	$\sum_A z_A$	Upper bound	Optimal size	Heuristic
(a) modem	6	37	32	20	31
(b) sample con.	6	612	60	34	57
(c) H.263 dec.	4	1190	2378	1257	1257
(d) FFT	11	94	992	504	808
(e) TDE	27	2867	7328	3680	5384



[3]Without lost of generality, tokens are assumed to have the same size.

VII. Related work

Few symbolic results about SDF graphs can be found in the literature. In this section, we present the most relevant ones.

Consistency can easily be checked analytically. The repetition vector can be computed symbolically as is it done in most dynamic parametric SDF models (*e.g.,* [2], [9]).

There is no exact analytical solution to check the liveness of a graph with buffers with fixed bounds. In [2], [3], the authors apply Eq. (1) transitively (which leads to nested ceilings) on edges of each cycle in the graph. Then, the obtained equations are linearized by over-approximating the ceiling function (*i.e.,* $\lceil x \rceil < x+1$). Yet, this is a conservative liveness analysis. As proved in [1], the minimum buffer size for which the simple graph $A \xrightarrow{p \quad q} B$ is live is equal to $p+q-\gcd(p,q)$[4]. This however does not imply that any graph whose channels are sized this way is live. Still, this analytical equation is used in many buffer sizing algorithms to compute a lower bound as a starting solution [3], [16].

Let $\vec{s}_i$ denotes the *token timestamp vector*, where each entry corresponds to the production time of tokens in the i^{th} iteration of the graph. Then, the max-plus algebra can be used to express the evolution of the token timestamp vector: $\vec{s}_i = M\vec{s}_{i-1}$. It has been proved that the eigenvalue of matrix M is equal to the period of the graph. In case of parametric rates, it is sometimes possible to extract a max-plus characterization of the graph with a parametric matrix [13]. However, this works only in cases where Eq. (1) can be somehow simplified to get rid of the ceiling function (*e.g.,* when p=1).

[10] presents a parametric throughput analysis for SDF graphs with *bounded* parametric execution times of actors but constant rates. Since rates and delays are non-parametric, the SDF-to-HSDF transformation is possible and the throughput analysis is based on the MCM. Therefore, all cycle means are linear functions in terms of parametric execution times. By using these linear functions, the parameter space is thus divided into a set of convex polyhedra called "throughput regions", each with a throughput expression. This approach has been extended in [7] to the case of scenario-aware dataflow (SADF) graphs.

A different analytic approach to estimate lower bounds of the maximum throughput is to compute strictly periodic schedules instead of ASAP schedules (*e.g.,* [5]). This approach is similar to our *Stretch* linearization method. The advantage of the strictly periodic scheduling approach is its capability to handle cyclic graphs. However, not all cyclic graphs have strictly periodic schedules (it depends on the number of initial tokens). Furthermore, experiments on real-life benchmarks show that these approaches result in huge over-approximations (sometimes 7 times the exact value) [5]. In theory, the over-approximation is not even bounded.

VIII. Conclusion

We have studied analytically the different cases of the execution of a completely parametric single edge dataflow graph. Then, we have presented the exact symbolic solutions for the minimum buffer size needed by a single edge to achieve its maximal throughput. Using these results and a linearization technique, we have provided safe upper bounds of buffer sizes of acyclic graphs for maximal throughput. Furthermore, we have proposed a heuristic to improve these bounds for graphs with a chain or a tree structure. Experimental results show that our heuristic improves the upper bound by 20% in average and can give the optimal solution for some real applications. We are following the same approach for exact and approximate symbolic evaluations of the latency of parametric graphs. Future work will concern the extension of these analysis to deal with general (*i.e.,* possibly cyclic) dataflow graphs.

[4]The equation is slightly different when there are initial tokens.

References

[1] S. S. Battacharyya, E. A. Lee, and P. K. Murthy. *Software synthesis from dataflow graphs.* Kluwer Academic Publishers, 1996.

[2] V. Bebelis, P. Fradet, A. Girault, and B. Lavigueur. BPDF: a statically analyzable dataflow model with integer and boolean parameters. In *Proceedings of the 11th ACM International Conference on Embedded Software*, pages 3:1–3:10, 2013.

[3] E. Bempelis. *Boolean Parametric Data Flow: Modeling - Analysis - Implementation.* PhD thesis, Université Grenoble Alpes, 2015.

[4] B. Bhattacharya and S. S. Bhattacharyya. Parameterized dataflow modeling of DSP systems. *Trans. Sig. Proc.*, 49(10):2408–2421, 2001.

[5] B. Bodin, A. Munier-Kordon, and B. de Dinechin. Periodic schedules for cyclo-static dataflow. In *Proceedings of the 11th Symposium on Embedded Systems for Real-time Multimedia*, pages 105–114, 2013.

[6] A. Bouakaz, P. Fradet, and A. Girault. Symbolic analysis of dataflow graphs (extended version). Technical Report 8742, INRIA, 2016.

[7] M. Damavandpeyma, S. Stuijk, M. Geilen, T. Basten, and H. Corporaal. Parametric throughput analysis of scenario-aware dataflow graphs. In *Proceedings of the 30th International Conference on Computer Design*, pages 219–226, 2012.

[8] K. Desnos, M. Pelcat, J. Nezan, S. S. Bhattacharyya, and S. Aridhi. PiMM: parametrized and interfaced dataflow meta-model for MPSoCs runtime reconfiguration . In *Proceedings of the 2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 41–48, 2013.

[9] P. Fradet, A. Girault, and P. Poplavko. SPDF: a schedulable parametric data-flow MoC. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 769–774, 2012.

[10] A. H. Ghamarian, M. C. W. Geilen, T. Basten, and S. Stuikj. Parametric throughput analysis of synchronous data flow graphs. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 116–121, 2008.

[11] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. In *Proceedings of the IEEE*, pages 1235–1245, 1987.

[12] O. Moreira, T. Basten, M. Geilen, and S. Stuijk. Buffer sizing for rate-optimal single-rate data-flow scheduling revisited. *IEEE Trans. Comput.*, pages 188–201, 2010.

[13] M. Skelin, M. Geilen, F. Catthoor, and S. Hendseth. Worst-cas throughput analysis for parametric rate and parametric actor execution time scenario-aware dataflow graphs. In *Proceedings of the 1st International Workshop on Synthesis of Continuous Parameters*, pages 65–79, 2014.

[14] S. Sriram and S. S. Bhattacharyya. *Embedded multiprocessors: scheduling and synchronization.* Marcel Dekker, Inc., 2000.

[15] S. Stuijk, T. Basten, M. Geilen, H. Corporaal, and M. Damavandpeyma. Throughput-constrained DVFS for scenario-aware dataflow graphs. In *Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium*, pages 175–184, 2013.

[16] S. Stuijk, M. Geilen, and T. Basten. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *Proceedings of the 43rd Annual Design Automation Conference*, pages 899–904, 2006.

[17] S. Stuijk, M. Geilen, and T. Basten. SDF^3: SDF for free. In *Proceedings of the 6th International Conference on Application of Concurrency to System Design*, pages 276–278, 2006.

[18] W. Thies and S. Amarasinghe. An empirical characterization of stream programs and its implications for languages and compiler design. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 365–376, 2010.