Scalable analysis for multi-scale dataflow models

Scalable Analysis for Multi-Scale Dataflow Models

HADI ALIZADEH ARA, Eindhoven University of Technology
AMIR BEHROUZIAN, Eindhoven University of Technology
MARTIJN HENDRIKS, ESI, TNO
MARC GEILEN, Eindhoven University of Technology
DIP GOSWAMI, Eindhoven University of Technology
TWAN BASTEN, Eindhoven University of Technology & ESI, TNO

Multi-scale dataflow models have actors acting at multiple granularity levels e.g. a dataflow model of a video processing application with operations on frame, line and pixel level. The state of the art timing analysis methods for both static and dynamic dataflow types aggregate the behaviours across all granularity levels into one, often large iteration, which is repeated without exploiting the structure within such an iteration. This poses scalability issues to dataflow analysis, because behaviour of the large iteration is analysed by some form of simulation which involves a large number of actor firings. We take a fresh perspective of what is happening inside the large iteration. We take advantage of the fact that the iteration is a sequence of smaller behaviours, each captured in a scenario, that are typically repeated many times. We use the (max, +) linear model of dataflow to represent each of the scenarios with a matrix. This allows a compositional worst-case throughput analysis of the repeated scenarios by raising the matrices to the power of the number of repetitions, which scales logarithmically with the number of repetitions, whereas the existing throughput analysis scales linearly. We moreover provide the first exact worst-case latency analysis for scenario-aware dataflow. This compositional latency analysis also scales logarithmically when applied to multi-scale dataflow models. We apply our new throughput and latency analysis to several realistic applications. The results confirm that our approach provides a fast and accurate analysis.

Additional Key Words and Phrases: dataflow, scenarios, (max,+), scalable analysis, throughput, latency

ACM Reference Format:
Hadi Alizadeh Ara, Amir Behrouzian, Martijn Hendriks, Marc Geilen, Dip Goswami, and Twan Basten. 2018. Scalable Analysis for Multi-Scale Dataflow Models. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (January 2018), 25 pages. https://doi.org/10.1145/3233183

1 INTRODUCTION

Assessing time-related performance is a crucial step in the design of many real-time systems. Throughput is often considered as one of the most important requirements when designing embedded software. The worst-case latency of a computation is often also important, for example, in control system design. Since there are usually trade-offs between such requirements and the

Authors' addresses: Hadi Alizadeh Ara, Eindhoven University of Technology, Eindhoven, The Netherlands, s.h.seyyed.alizadeh@tue.nl; Amir Behrouzian, Eindhoven University of Technology, Eindhoven, The Netherlands, A.R.Baghban.Behrouzian@tue.nl; Martijn Hendriks, ESI, TNO, Eindhoven, The Netherlands, martijn.hendriks@tno.nl; Marc Geilen, Eindhoven University of Technology, Eindhoven, The Netherlands, M.C.W.Geilen@tue.nl; Dip Goswami, Eindhoven University of Technology, Eindhoven, The Netherlands, D.Goswami@tue.nl; Twan Basten, Eindhoven University of Technology , & ESI, TNO, Eindhoven, The Netherlands, A.A.Basten@tue.nl.

1539-9087/2018/1-ART1 $15.00
https://doi.org/10.1145/3233183

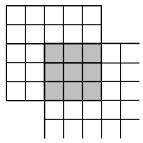


Fig. 1. A 5x5 convolution filter.

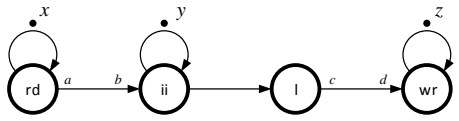
amounts of resources allocated to the system, there is a need for analysis techniques that are not only fast, to explore the trade-off spaces in a reasonable time, but also accurate to avoid over-allocation of the resources.

Model-based timing analysis techniques are used to ensure performance for such systems. Timed dataflow is a task-level model of computation used in a number of model-based design approaches for real-time parallel streaming applications [4, 23]. Static dataflow such as Synchronous Dataflow (SDF) [20] and variants of dynamic dataflow with good analysability properties such as Cyclo-Static Dataflow (CSDF) [5] and Finite-State-Machine Scenario-Aware Dataflow (FSM-SADF) [32] are popular among resource allocation and buffer sizing algorithms [21, 31].

Analysis methods for dataflow models often involve a simulation step where the tasks start their execution as soon as their dependencies are met. Then the completion times of the executions are captured for analysis. For the techniques that are based on operational semantics [15, 28, 29] the simulation continues until the periodic phase of the dataflow behaviour is found. For the techniques based on (max, +) algebra [2, 13] the simulation phase involves the task executions that are required to generate the so-called timing matrix and the final analysis is a spectral analysis on the matrix. For many practical applications, the number of task executions within the simulation is very large.

As an example, consider a convolution filter applied to frames of streaming video. The filter uses a kernel of 5 by 5 pixels and applies padding at the border of the frame (Fig. 1). We assume that a single frame is an image with a width of W pixels and a height of H pixels. The video stream is delivered to the filter pixel-by-pixel, pixels being ordered frame-by-frame, line-by-line – within a frame, from top line to bottom line and from left pixel to right pixel. The filter produces filtered video with pixels in the same order. The filter shows a specific pattern of data dependencies (it needs to collect the required input data for the convolution kernel) depending on the location of the kernel. Initially the centre of the kernel is located on the top left pixel of the frame (Fig. 1). The kernel slides to the right, pixel by pixel, for a whole line and then starts form the left side of the next line. This continues until the centre of the kernel reaches the bottom right pixel of the frame. When the kernel is on the initial location, it needs to have read 9 input pixels (kernel elements with gray colour in Fig. 1) and use border padding for the elements that fall out of the image (kernel elements with diagonal pattern in Fig 1). Since the input data is ordered line-by-line, the filter needs to read 2 lines and 3 pixels from the input buffer before it can carry out the first convolution computation. Then the kernel slides one pixel to the right and it needs to read only one extra input pixel to do the second convolution computation. We summarize the behaviour of the filter in terms of reading input pixels and producing output pixels by the following pattern of phases.

(1) **frame rush-in phase**: read pixels one by one without producing any output yet for 2 lines of the image, or $2W$ pixels;
(2) **line rush-in phase**: read two pixels without producing output;
(3) **line computation phase**: one pixel is read and one pixel is produced for a whole line minus two pixels: $W - 2$ pixels;



Scenario	Rates				Execution Times				Repetition Counts			
	a	b	c	d	rd	ii	l	wr	rd	ii	l	wr
ri	1	1	0	0	2	2	1	0	1	1	1	0
cm	1	1	1	1	2	4	5	2	1	1	1	1
ro	0	0	1	1	0	3	4	2	0	1	1	1

$$\mathbf{G}_{ri} = \begin{bmatrix} 2 & -\infty & -\infty \\ 4 & 2 & -\infty \\ -\infty & -\infty & 0 \end{bmatrix}$$

$$\mathbf{G}_{cm} = \begin{bmatrix} 2 & -\infty & -\infty \\ 6 & 4 & -\infty \\ 13 & 11 & 2 \end{bmatrix}$$

$$\mathbf{G}_{ro} = \begin{bmatrix} 0 & -\infty & -\infty \\ -\infty & 3 & -\infty \\ -\infty & 9 & 2 \end{bmatrix}$$

Fig. 2. SDF models and the timing matrices of the convolution filter modes.

(4) **line rush-out phase**: two more pixels are produced without reading new input (using padding);
(5) repeat steps 2–4 for $H - 2$ lines;
(6) **frame rush-out phase**: produce two more lines of output without new input (using padding): $2W$ pixels.

This phase pattern can be described by a repetitive sequence, composed of three different data dependency modes. Let *ri* (rush-in) denote a mode in which one input pixel is read but no output pixel is produced, *cm* (computation) be a mode that reads one input pixel and produces one output pixel and finally, *ro* (rush-out) denote a mode in which no input is read but one output is produced. Each of the phases in the pattern is composed of a constant repetition of one of theses modes. For instance the frame rush-in phase is composed of $2W$ repetitions of mode *ri*. This phase can be represented in a compact way by expression $(ri)^{2W}$. By representing the rest of the phases in a similar way, the repeated phase pattern can be described by the following expression, where ω indicates infinite repetition corresponding to an infinite stream of input frames.

$$\left((ri)^{2W}\left((ri)^2(cm)^{W-2}(ro)^2\right)^{H-2}(ro)^{2W}\right)^{\omega}$$

When analysing the frame-level behaviour of the convolution filter with the state of the art CSDF [5] and FSM-SADF [13] analyses, a simulation that goes through all variations in the phase pattern is needed to obtain the timing matrix. For a 2048×2048 frame, this simulation contains 1.67e+7 firings (3 firings for every pixel in the rush-in and rush-out phases and 4 firings for every pixel in the computation phase). The run-time of a timing analysis that evokes such a simulation on today's computers is in the order of seconds. However, this is still too much for trade-off analysis methods such as buffer sizing, since the timing analysis techniques might be called more than 300 thousand times during the analysis [31]. It is known that a conservative approximation can be used to reduce the number of firings, for instance by aggregating the behaviour of multiple executions into a single vectorized execution [25]. However, this may lead to pessimistic results. We observe that a more efficient analysis is possible by using the mode information.

Since the modes in the convolution filter correspond to fixed data dependency relations, they can be described using SDF graphs. For instance, we use the graph shown in Fig. 2 to describe the modes in the filter. In this figure, nodes represent actors. The rd actor models reading of the input pixels. The ii and l actors model the starting of the convolution computations or the *initiation interval* and computation *latency* respectively. The wr actor models the production and writing of the output pixels. Actor firings correspond to task executions and they take some time (which is

shown for every mode, in the table in Fig. 2). The directed edges in the graph model channels. In mode *cm*, the channel from rd to ii ensures that the convolution computations start only after the reading is complete. In this mode, when rd completes its firing, it produces a token on this channel. This token is then consumed by ii when it starts its firing. The number of tokens consumed and produced by actors from channels are called actor rates. For instance, in mode *cm*, on the channel from rd to ii, rd has a production rate of 1 and ii has a consumption rate of 1. The table shows the actor rates that are different in every mode (the rest of the rates are equal to 1 in every mode). Channels may initially contain a number of tokens. A channel from an actor to itself, i.e. a self-edge, with an initial token on it, ensures that there will be only one execution of that actor at a time.

We observe that a simulation that involves 10 actor firings suffices to analyse the timing of all modes in the filter. Every mode in the filter defines a set of actor firings in the SDF graph depending on the behaviour it represents. For instance, mode *cm* involves one firing from every actor in the graph, as it represents reading an input, performing a computation and, writing an output. The table in Fig. 2, shows the number of firings (repetition counts) of every actor in every mode. The essential timing information of each mode can be captured by simulating the execution of the mode, i.e., by simulating all actor firings in the mode. During the simulation, the tokens are assigned with time-stamps that indicate their earliest availability times on the channels. After a mode executes, the tokens that are left on the channels are called final tokens. In the filter, execution of every mode leaves one token on every self-edge in the graph.

The timing matrix [13] of a mode captures the essential timing relations between the time-stamps of initial and final tokens. To specify theses relations, let x, y and z label the initial and final tokens on the self-edges of actors rd, ii and wr, respectively. The timing matrix of every mode is shown in Fig 2. In these matrices, the timing relations of tokens labelled x, y and z are captured in the first, second and third column/row of the matrices, respectively. For example, for *cm*, the third row of the matrix indicates that there are at least 13, 11 and 2 time units difference between the time-stamps of initial tokens x, y and z and the time-stamp of final token z after *cm* is executed. $-\infty$ indicates that there is no dependency relation. In a similar way, the timing information of modes *ri* and *ro* is captured after 3 actor firings each (modes *ri* and *ro* exclude the firing of wr and rd, respectively).

Using the timing matrices, the frame-level behaviour that is described as the above repetitive pattern of modes can be compositionally analysed. Given the time-stamps of the initial tokens in a mode, we can compute the time-stamps of the final tokens using a matrix-vector multiplication in (max, +) algebra. In this algebra, addition and max operator respectively take the role of multiplication and addition of traditional linear algebra. In the filter, if we collect the time-stamps of the initial tokens of a mode m in a vector $\boldsymbol{\gamma} = [\ t_x \quad t_y \quad t_z\ ]^T$, a vector that corresponds to the time-stamps of the final tokens is computed as $\mathbf{G}_m\boldsymbol{\gamma}$.

The throughput of the filter can be obtained by analysing the growth rate of the time-stamp vector after execution of every frame. Given the initial time-stamp vector in the first mode of the pattern, it is multiplied by its matrix to compute the time-stamp vector of the final tokens in the first mode. Then the computed time-stamp vector is used as the initial time-stamp vector in the second mode of the pattern. Thus, it is multiplied by the matrix of the second mode to compute the initial time-stamp vector of the third mode and so on. This way, the time-stamp vector after execution of every frame can be computed and its growth rate can be analysed.

Alternatively, we can define a matrix that abstracts the execution of the model for production of a full frame. This matrix, called the frame-level matrix, abstracts all actor firings during the production of every individual pixel of the frame, in the relation between the time-stamps of the initial tokens in the first mode of the pattern and the final tokens in the last mode of the frame pattern. The frame level-matrix can be used for analysis. For example, throughput is obtained through the standard spectral analysis on this matrix. The frame-level matrix can be computed by

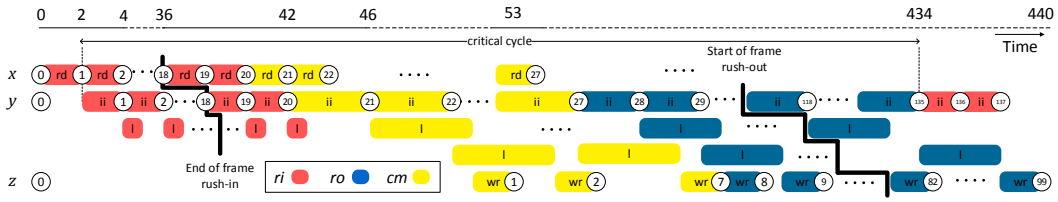


Fig. 3. Execution of the convolution filter on a 9×11 frame.

multiplying the matrices of modes in the order that is the reverse of the appearance order of their corresponding modes in the frame pattern. Let's consider a small 9×11 frame ($W = 9, H = 11$) to be able to show the execution traces of the filter. The frame-level matrix is defined as follows.

$$\mathbf{G}_{frame} = \mathbf{G}_{ro}^{2W} \left(\mathbf{G}_{ro}^{2} \mathbf{G}_{cm}^{W-2} \mathbf{G}_{ri}^{2}\right)^{H-2} \mathbf{G}_{ri}^{2W} = \mathbf{G}_{ro}^{18} \left(\mathbf{G}_{ro}^{2} \mathbf{G}_{cm}^{7} \mathbf{G}_{ri}^{2}\right)^{9} \mathbf{G}_{ri}^{18} = \begin{bmatrix} 198 & -\infty & -\infty \\ 434 & 432 & -\infty \\ 440 & 438 & 198 \end{bmatrix}.$$

Spectral analysis on this matrix reveals that the growth rate of the time-stamp vectors produced by indefinite repetition of the frame pattern is 432 time units per frame and it has a critical dependency on the second column of the second row of $\mathbf{G}_{frame}$. This element corresponds to the time difference between the availability time of the token on the self-edge of actor ii before and after the production of a frame. This critical dependency is also observed in the execution trace of the filter in Fig. 3. Hence, the throughout is equal to $99/432$ pixels per time unit or $1/432$ frames per time unit.

The matrix $\mathbf{G}_{frame}$ can be efficiently computed. If a mode (or a pattern of modes) corresponds to a matrix $\mathbf{G}$, then n repetitions of that mode correspond to the matrix $\mathbf{G}^n$, which can be computed from $\mathbf{G}$, in $\mathcal{O}(\log n)$ time. Observe that, this way, we can analyse a fixed number of repetitions of a mode in time logarithmic in the number of repetitions (instead of linear in the number of actor firings as in the state of the art exact methods).

The convolution filter is an example of a multi-scale application for which the behaviour under analysis involves different sub-behaviours that occur at multiple granularity levels. This can be recognized in many other real-life examples, such as applications that include memory transactions with large blocks of memory that are communicated through a network-on-chip that operates at the level of much smaller individual flits [24]. An H.263 decoder is another example. A dataflow model of this application is provided in [28]. The decoder accepts video streams of frames as inputs, decomposes them into small blocks of data items called macro blocks and finally, reconstructs the decoded frame from macro blocks. As a result, the behaviour of the decoder involves few task executions at the frame level but many task executions at the macro block level.

The contribution of this work is threefold. The first novelty is the representation of the multi-scale behaviour of a dataflow application as a regular expression built from execution modes, as illustrated above. This results in an FSM-SADF model in which each mode is represented by a scenario SDF model and the regular language of mode sequences is represented as an *ω-regular expression*, which enables efficient timing analysis.

As the second contribution, we provide an algorithm for exact worst-case throughput analysis of FSM-SADF where the scenario sequences are given by a regular expression. Given such an FSM-SADF, the algorithm first tries to transform it into a new FSM-SADF with fewer Finite State Machine (FSM) states. This is done by finding sub-expressions formalizing sets of scenario sequences, the behaviour of which can be abstracted by timing matrices in line with the convolution example. Without losing accuracy, the scenarios of the resultant FSM-SADF are defined to abstract these sets

of scenario sequences and a new regular expression is constructed using the defined scenarios. The final analysis invokes the existing FSM-SADF analysis [11] with a compact FSM that recognizes the language formalized by the new expression. We prove that the resultant FSM-SADF has the same worst-case throughput as the original FSM-SADF. For an SDF or a CSDF model that is represented by an indefinite repetition of a sequence of scenarios, our throughput analysis scales linearly in the length of the *representation* of the sequence as a regular expression (instead of the length of the sequence itself). Moreover, for an important class of general FSM-SADF (i.e. when the expression recognizes a set of scenario sequences instead of a single sequence), the throughput analysis is polynomial in the length of the expression.

As the final contribution, we provide an algorithm to compute the exact latency of an FSM-SADF with a given regular expression. The algorithm computes a vector that expresses the worst-case timing relations between the time-stamps of initial tokens and latencies of outputs among all possible scenario sequences. The algorithm computes this vector in a recursive fashion following the regular expression by analysing every type of syntactic composition. The latency is equal to the inner product of this vector and the vector that contains the time-stamps of initial tokens. We show that the latency vector computation time scales linearly in the length of the regular expression.

The overview of the paper is as follows. Section 2 discusses the works related to the throughput and latency analysis of dataflow graphs. Section 3 summarizes the necessary (max, +) and dataflow background. Our throughput and latency analysis methods are explained in Sections 4 and 5. Section 6 presents experiments and provides a discussion on the results. Section 7 concludes.

2 RELATED WORK

The prime performance property of interest for dataflow models of computation is throughput. The earliest works on throughput analysis of SDF [20, 27] use a conversion of the graph to its equivalent Homogeneous Synchronous Dataflow (HSDF) graph. HSDF is the single-rate version of SDF. This enables the use of Maximum Cycle Mean (MCM) or Maximum Cycle Ratio (MCR) [16] analysis techniques to obtain the throughput of the graph. This method can also be applied to CSDF by first using a conversion to HSDF [8]. Apart from the fact that the conversion step itself is time consuming, such conversions often result in a very large HSDF (especially for CSDF graphs) which poses scalability issues even for efficient MCM analysis techniques such as Karp's algorithm [18]. Approaches to provide an accurate but more scalable throughput analysis for SDF and CSDF are provided by Ghamarian et al. and Stuijk et al., respectively [14, 31]. In both cases, the analysis can be directly applied to the original graph, which removes the costly conversion step of the earlier works. They both include a simulation phase based on the operational semantics of dataflow. It still simulates all actor firings in each iteration, but stores only one state per iteration. The algorithm builds a global state-space representation of the self-timed execution of the dataflow graph by the simulation. It is known that the self-timed execution of a consistent and strongly-connected (C)SDF consists of a transient phase followed by a periodic phase in which actors fire in a periodic fashion. Throughput is extracted from the periodic phase of the self-timed execution–the transient phase is often very short in practice, but can be arbitrarily large in the worst case.

Theelen et al. provide a throughput analysis for Scenario-Aware Dataflow (SADF) [33]. The analysis is built upon a state-space representation of the graph. The state-space representation captures the behaviour of the graph across sequences of scenarios. Since transitions are at the level of individual firings of actors, the resulting state space becomes very large with larger models and leads to tractability issues. The FSM-SADF analysis [11, 13] uses a symbolic simulation method to get the (max, +) representation of each scenario. This representation abstracts the actor firing dependencies within the scenarios. Then such a representation together with the FSM is used to generate either a state-space of all reachable states (time-stamp vectors) or a (max, +) automaton.

In the analysis techniques for FSM-SADF, a (partial) iteration of the graph in a particular scenario is captured in a single transition of the state-space or automaton, instead of the individual actor firings. This leads to improved scalability. The analysis is ultimately mapped on a MCR analysis on a directed graph of the state-space or directly on the automaton. Irrespective of the size of the resulting directed graph or the automaton and the complexity of the final MCR analysis, the process of generating the state space of the time-stamp vectors or the automaton scales linearly in the number of FSM states. We use the same symbolic simulation method to obtain (max,+) representations for scenarios as Geilen et al. [12, 13]. Then we use a transformation on the given FSM-SADF to generate a new FSM-SADF with fewer number of FSM states, leading to further scalability improvements for multi-scale dataflow models compared to the analyses of Geilen et al. [12, 13].

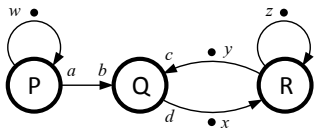
Besides exact throughput analysis, it is possible to approximate throughput. An approximate conversion method [8] can be used to generate pessimistic and optimistic HSDF abstractions of an SDF, but with the same size as the original graph, the same number of actors and edges. Throughput analysis over the approximate HSDF is done for the sake of shorter analysis run-time. We compare our technique to the conservative approximate method of De Groote et al. [8] in Section 6. Our technique turns out to be better scalable, despite being exact.

A second performance property of interest is latency. Geilen et al. provide a definition for latency which we also use as a basis in this paper (where it is in fact generalized) [13]. They sketch a possible way to compute the latency for FSM-SADF without providing an algorithm. The proposed approach requires the generation of the state-space which scales linearly in the number of FSM states. Moreira et al. [22] use the same latency notion as Geilen et al. [13]. They provide an algorithm to compute the latency. Their approach promotes the use of static periodic schedules as a conservative approximation of self-timed schedules. Therefore, the analysis gives an upper bound on the latency. Moreover, it can only be applied to SDF, CSDF, i.e., the deterministic cases of FSM-SADF where the FSM accepts one scenario sequence. We provide an algorithm for latency analysis of an arbitrary FSM-SADF model. We show that when we represent the set of possible scenario sequences of an FSM-SADF with a regular expression, the latency computation time using our algorithm scales linearly in the length of the expression.

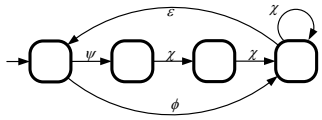
3 DATAFLOW AND (MAX,+) PRELIMINARIES

3.1 Synchronous Dataflow

In SDF models, tasks are abstracted by *actors*. Applications are modelled by graphs, where the nodes are the actors. Fig. 4a shows an example SDF graph. A *channel* from actor P to actor Q models a firing dependency of Q on P. The dependencies between actor *firings* are captured by a token consumption and production mechanism. When all input channels of an actor contain sufficient tokens, an actor is enabled and may *fire* (execute). Actor firings produce tokens on the channels; tokens produced by actor firings are consumed by (other) actors that become enabled; the enabled actors fire and produce new tokens and so on. This way compact dataflow models can capture complex task dependencies. Channels may have *initial tokens*, indicated by black dots near channels in the graph. The tokens in Fig. 4a are labelled (w, x, y, z) for the purpose of referencing. In SDF actors consume and produce a constant integer number of tokens from their input channels and output channels respectively, which are determined by the actor *rates* on the channels. Channel (P, Q) in Fig. 4a has rates a and b, for example. Rates equal to 1 are not shown in SDF graphs. Executing actors typically takes time. In SDF, the execution time of an actor is constant for all its firings. An *execution* of an SDF graph is a finite or infinite sequence of actor firings. In a *self-timed execution*, actors fire as soon as they are enabled.



(a) SDF graph structure



(b) FSM

Scenario	Rates				Exe. Times			Rep. Counts			No. I. and F. Tokens					Output
	a	b	c	d	P	Q	R	P	Q	R	(P, P)	(P, Q)	(Q, R)	(R, Q)	(R, R)	
ϕ	1	1	2	2	4	2	2	1	1	2	1,1	0,0	1,1	1,1	1,1	R
χ	0	0	1	1	0	1	4	0	1	1	1,1	0,0	1,1	1,1	1,1	R
ψ	0	0	0	0	2	0	0	1	0	0	1,1	0,0	1,1	1,1	1,1	R

Fig. 4. An example FSM-SADF.

3.2 Synchronous Dataflow Scenarios

SADF is a variant of dataflow that introduces a form of dynamism in dataflow behaviour. It defines a set of SDF scenarios and a suitable formalism to express possible sequences of scenarios. An SDF scenario defines a finite non-empty set of actor firings. The set is specified by the exact number of times that each actor in the graph fires (also called the actor *repetition counts*). A scenario is typically defined by means of an SDF graph. There are three example scenarios in Fig. 4 namely ϕ, χ and ψ. All example scenarios have the same SDF graph structure in this example, but different parameters i.e. actor rates and execution times. The table beneath the SDF graph contains the graph parameters and repetition counts for all scenarios.

A scenario execution refers to the execution of all actor firings specified by the scenario. Execution of a scenario may leave a number of tokens on the channels, called *final tokens*. An SADF executes scenarios one after another. To express task dependencies among scenario executions, a scenario labels a number of its initial and final tokens. In an SADF, the timing information from one scenario to the next is conveyed from the labelled final tokens in the scenario to the initial tokens with the same labels in the next scenario. The table in Fig. 4 contains the number and location (the channel) of the initial and final tokens for every scenario. Letters w, x, y and z are used to label initial and final tokens in all of the three scenarios as shown in the SDF graph structure (Fig. 4a). A transition from scenario χ to ψ is *consistent*, for example, since the number of final tokens in χ and their labels match the number of initial tokens and their labels in ψ.

SADF are often analysed to provide performance guarantees on non-terminating runs of the applications. For example, when used to model streaming applications, the analysis is done for an infinite stream of data. Therefore the timing behaviour of an SADF is defined for an infinite sequence of scenarios, assuming that all scenario transitions in the sequence are consistent. In general SADF allows for non-deterministic scenario transitions, leading to an infinite number of possible scenario sequences. For instance, for an FSM-SADF, scenario transitions are captured by an FSM. Fig. 4b shows an example FSM, where the transitions are labelled with scenarios except for the ϵ-transition, which denotes an empty transition (as standard in non-deterministic automata). In a consistent FSM-SADF, all scenario transitions in all scenario sequences accepted by the FSM are consistent. For analysis purposes, we associate output with some actor firings in a scenario. In the example FSM-SADF, we associate output with firing of actor R in all scenario graphs (Output in the table of Fig. 4). For instance ϕ produces two outputs during execution, since there are two firings of R in that scenario.

3.3 (max,+) Representation of Scenarios

We first introduce (max, +) algebra notations (See [3] for background on (max, +) algebra). The (max, +) algebraic addition and multiplication are denoted by $\oplus$ and $\otimes$ respectively: $a \oplus b = \max\{a, b\}$ and $a \otimes b = a + b$ for $a, b \in \mathbb{R}_{-\infty} ::= \mathbb{R} \cup \{-\infty\}$. This algebra operates on real-valued numbers extended with $-\infty$. Multiplication and addition on matrices, matrix-vector multiplication and inner product of vectors have the same concepts as their counterparts in ordinary algebra, but with the use of (max, +) algebraic operations. Let $\mathbf{A}$,$\mathbf{B}$ and $\boldsymbol{v}$ be two matrices and a vector respectively. For readability we write $\mathbf{AB}$ and $\mathbf{A}\boldsymbol{v}$ instead of $\mathbf{A} \otimes \mathbf{B}$ and $\mathbf{A} \otimes \boldsymbol{v}$. Let $n \in \mathbb{N} \cup \{0\}$. $\mathbf{A}^n$ denotes raising $\mathbf{A}$ to the power of n. $\mathbf{A}^0 = \mathcal{J}$. $\mathcal{J}$ is the identity matrix in (max, +) algebra (the diagonal elements are 0 and the other elements are $-\infty$). $\mathbf{0}$ denotes a column vector of appropriate size in which all entities are 0. $\bigoplus$ is the summation quantifier in (max, +) algebra. Summation over an empty set equals $-\infty$. The star closure of a square matrix $\mathbf{M}$ is a matrix $\mathbf{M}^* = \bigoplus_{k \geq 0} \mathbf{M}^k$. Operation $\oplus$ on matrices corresponds to point wise $\oplus$ operations between the corresponding elements of the matrices. $\mathbf{M}^*$ exists if and only if there are no positive cycles in the adjacency graph of $\mathbf{M}$ i.e. if $\mathbf{M}$ has no positive eigenvalues. Subtraction of a scalar c from a matrix $\mathbf{M}$, i.e. $\mathbf{M} - c$ or $\mathbf{M} \otimes (-c)$, is to subtract c from all elements of $\mathbf{M}$. The norm of a matrix $\|\mathbf{M}\|$ or a vector $\|\boldsymbol{v}\|$ is equal to the maximum entry in the matrix or the vector. If $\boldsymbol{v}$ is a vector, $\boldsymbol{v}_k$ denotes the k^{th} element of the vector. Let $\mathbf{A} = [a_{ij}]$ and $\mathbf{B} = [b_{ij}]$ be matrices of the same size, where a_{ij} and b_{ij} denote the element on row i, column j of their corresponding matrices. We write $\mathbf{A} \leq \mathbf{B}$ to denote that for every i and j, $a_{ij} \leq b_{ij}$. Let $\mathbf{C}$ be a matrix of appropriate size, $\mathbf{A} \leq \mathbf{B}$ implies that $\mathbf{AC} \leq \mathbf{BC}$, $\mathbf{CA} \leq \mathbf{CB}$ and $\mathbf{A} \oplus \mathbf{C} \leq \mathbf{B} \oplus \mathbf{C}$.

The (max, +) representation of a scenario is a mathematical expression of the relation between the production times of the labelled final tokens and the availability times of the labelled initial tokens in the self-timed execution of the scenario. For a scenario s, if we collect the availability times of the labelled initial tokens in vector $\boldsymbol{\gamma}$ and the production times of the labelled final tokens in vector $\boldsymbol{\gamma}'$, this relation can be expressed as the following equation in (max, +) algebra [13].

$$\boldsymbol{\gamma}' = \mathbf{G}_s \boldsymbol{\gamma} \tag{1}$$

Eq. 1 is in fact a compact form of a set of linear equations in (max, +) algebra. We refer to $\mathbf{G}_s$ as the *state matrix* (also called scenario matrix) of s and $\boldsymbol{\gamma}$ is the *state vector* (also called dater function or time-stamp vector). For instance, in Fig. 4, we can collect the time-stamps of the labelled initial tokens in vector $\boldsymbol{\gamma} = [\ t_w \quad t_x \quad t_y \quad t_z\ ]^T$ and final tokens in vector $\boldsymbol{\gamma}' = [\ t'_w \quad t'_x \quad t'_y \quad t'_z\ ]^T$ and describe scenario ϕ by the following matrix. A systematic way of generating (max, +) representations of dataflow scenarios can be found in [26].

$$\mathbf{G}_\phi = \begin{bmatrix} 4 & -\infty & -\infty & -\infty \\ 6 & 4 & 2 & 4 \\ 8 & 6 & 4 & 6 \\ 8 & 6 & 4 & 6 \end{bmatrix}$$

An entry t at column k and row m in $\mathbf{G}_s$ specifies that there is *at least* a time difference of t time units between the availability of token k before the execution of s and the production of token m after s is executed. For instance the entry 6 on the first column of the second row of $\mathbf{G}_\phi$ implies that the production of final token x happens at least 6 time units after initial token w becomes available. This is a valid statement since the dependency path from w to x goes through one firing of P (with execution time of 4) and one firing of Q (with execution time of 2); therefore there is at least $4 + 2 = 6$ time units difference between the availability of w and the production of x. An entry $-\infty$ implies that there is no dependency relation.

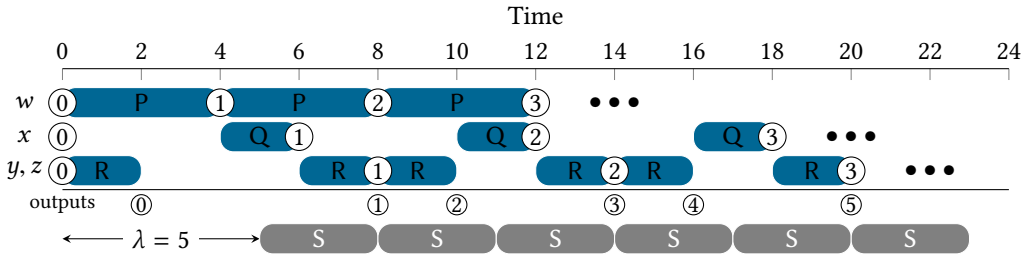


Fig. 5. Self-timed execution of scenario sequence ϕ^ω in Fig. 4.

We can use $\mathbf{G}_s$ to determine the evolution of state vectors for any given scenario sequence. For instance, assuming that the initial tokens are available at time-stamp $\boldsymbol{\gamma}_0 = \mathbf{0}$, we can find the production times of final tokens after the execution of ϕ as:

$$\boldsymbol{\gamma}_1 = \begin{bmatrix} 4 & -\infty & -\infty & -\infty \\ 6 & 4 & 2 & 4 \\ 8 & 6 & 4 & 6 \\ 8 & 6 & 4 & 6 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 4 \otimes 0 \oplus -\infty \otimes 0 \oplus -\infty \otimes 0 \oplus -\infty \otimes 0 \\ 6 \otimes 0 \oplus 4 \otimes 0 \oplus 2 \otimes 0 \oplus 4 \otimes 0 \\ 8 \otimes 0 \oplus 6 \otimes 0 \oplus 4 \otimes 0 \oplus 6 \otimes 0 \\ 8 \otimes 0 \oplus 6 \otimes 0 \oplus 4 \otimes 0 \oplus 6 \otimes 0 \end{bmatrix} = \begin{bmatrix} 4 \\ 6 \\ 8 \\ 8 \end{bmatrix}.$$

The new state vector $\boldsymbol{\gamma}_1$ signifies that during the execution of scenario ϕ (perhaps as the first scenario in a scenario sequence), the final tokens will be produced at times [4,6,8,8]. Let ϕ^ω denote indefinite repetition of scenario ϕ. Fig. 5 shows the self-timed execution of ϕ^ω. The initial tokens are shown by the big nodes with label zero. Number k inside the other big nodes shows the production of final tokens after the execution of the k^{th} scenario in the sequence. Note that these tokens are also the initial tokens for the next scenario in the sequence and the state vector $\boldsymbol{\gamma}_2$ can be computed from the state matrix of the next scenario by using Eq. 1 when $\boldsymbol{\gamma}$ is substituted by $\boldsymbol{\gamma}_1$ and so on. For example for ϕ^ω we can compute the k^{th} state vector using the equation below.

$$\boldsymbol{\gamma}_k = \mathbf{G}_\phi \boldsymbol{\gamma}_{k-1} = \mathbf{G}_\phi^k \boldsymbol{\gamma}_0.$$

For analysis purposes, we need to capture the production times of outputs produced during a sequence of scenarios. The output production times might be directly accessible through the state vectors. For example in Fig. 5 it is observed that the production of every other output is synchronized with the production of the final tokens y and z i.e. it is reflected in the third and fourth elements of state vectors. However, this is not always the case. In general, given a state vector $\boldsymbol{\gamma}$, the output production times in a scenario s can be obtained using a matrix-vector multiplication $\mathbf{H}_s\boldsymbol{\gamma}$, similar to the one introduced in [1]. We refer to $\mathbf{H}_s$ as the *output matrix* of the scenario s. It has as many rows as there are outputs in s. Each row of this matrix expresses the relation between the state vector and the production time of an output in s as the inner product of each row and the state vector. We assume the rows are ordered such that the first row corresponds to the first output. For scenario ϕ, the output matrix is defined as follows.

$$\mathbf{H}_\phi = \begin{bmatrix} -\infty & 2 & -\infty & 2 \\ 8 & 6 & 4 & 6 \end{bmatrix}.$$

For example, using $\mathbf{H}_\phi$ the output production times for ϕ^ω can be computed as follows.

$$\begin{bmatrix} p_{2k} & p_{2k+1} \end{bmatrix}^T = \mathbf{H}_\phi \boldsymbol{\gamma}_k = \mathbf{H}_\phi \mathbf{G}_\phi^k \boldsymbol{\gamma}_0.$$

p_k is the time instance at which the k^{th} output is produced. In Fig. 5 the production time of outputs are shown by small nodes labelled with k.

3.4 FSM-SADF

This section provides a formal definition of FSM-SADF and defines the performance metrics we provide analysis for. The definition of FSM-SADF is adapted from the definition provided by Geilen et al. [11]. An FSM-SADF is defined by a tuple (S, g, r, i, f, o, L). The set S contains a finite number of scenarios. Every scenario $s \in S$ has an associated SDF graph $g(s)$ and a vector $r(s)$ of actor repetition counts. The function $r(s)$ maps every actor of $g(s)$ to a non-negative number. The graph $g(s)$ has a number of initial tokens $i(s) \in \mathbb{N}$ distributed over its edges. After scenario s executes, it leaves a number of final tokens $f(s) \in \mathbb{N}$ on some edges of $g(s)$. We assume initial and final tokens with the same label to be coupled. For every graph $g(s)$, we can construct the state matrix $\mathbf{G}_s \in \mathbb{R}_{-\infty}^{f(s)\times i(s)}$. The language L describes a set of infinite scenario sequences. We use L in the definition of FSM-SADF instead of using a specific representation of it such as an FSM, since we use different representations of the language in this paper, FSMs and regular expressions.

The throughput of dataflow models is determined by how often an actor fires or how often an actor produces tokens during an infinite execution of the model. We quantify the throughput of a given scenario sequence of an SADF by the average number of outputs produced per time unit during the execution of that sequence. The function $o : s \rightarrow \mathbb{N} \cup \{0\}$ maps each scenario to a non-negative number that corresponds to the number of outputs produced in that scenario. Recall that in Fig. 4 we associated output with firing of R. That is, we have $o(\psi) = 0$, $o(\chi) = 1$ and $o(\phi) = 2$.

A formal definition for the throughput of a given FSM-SADF scenario sequence is as follows.

$$\tau(\bar{s}) = \lim_{i\rightarrow\infty} \sup \frac{\Sigma_{n=1}^{i} o(s_n)}{\left\|\boldsymbol{\gamma}_i\right\|}. \tag{2}$$

The throughput of ϕ^ω is 1/3. This is also observed from the execution trace in Fig. 5. Actor R fires and produces an output once every three time units on average. The worst-case throughput of an FSM-SADF is obtained among the set of all sequences (words) in the language L, as follows.

$$\tau = \inf_{\bar{s}\in L} \tau(\bar{s}). \tag{3}$$

A throughput analysis based on this definition is provided by Geilen et al. [11]. The throughput in the example of Fig. 4 is 0.25.

There are different possible definitions for latency in the literature. We focus on a definition which is a general version of the one given by Geilen et al. [13]. We assume the outputs produced during the execution of a scenario sequence will be consumed by an external actor S that is *periodically scheduled* with period μ (it is not self-timed). Every firing of this actor consumes an output and its execution trace is shown in Fig. 5. For the external actor to have a feasible schedule, an output should be produced no later than the start time of the external actor for all firings. If the throughput is not lower than $1/\mu$, there exists a feasible periodic schedule for the external actor. That is, there exists a λ such that for all k it holds that $p_k \leq \lambda + k\mu$. Recall that we can compute the production times of outputs produced during a scenario s using the output matrix $\mathbf{H}_s \in \mathbb{R}_{-\infty}^{o(s)\times i(s)}$. We define the latency of a scenario sequence relative to a period μ as the earliest start time of the first firing of the periodically scheduled external actor that consumes outputs with period μ. This means the latency is defined as the smallest λ such that the above inequality holds for all k. Given an initial state $\boldsymbol{\gamma}_0$, the latency of a scenario sequence $\bar{s}$ relative to period μ is defined as follows.

$$\lambda(\bar{s}, \boldsymbol{\gamma}_0, \mu) = \max_{k\geq 0} p_k - \mu k \tag{4}$$

According to Fig. 5, given initial state $\boldsymbol{\gamma}_0 = \mathbf{0}$, the latency of ϕ^ω relative to period 3 is 5. Considering the production times of outputs and the firing start times of the external actor S, the first start time of S cannot happen earlier than 5. Since the second firing of S requires output number one which is produced at 8 the earliest. Geilen et al. provide a similar latency definition except that there is no notion of output actors and an execution of any scenario is regarded as production of one output [13]. Moreira et al. define the latency as the maximum time interval between the firings of a sink and an a source actor for all iterations of a CSDF graph [22]. In case that the source is periodically scheduled with period μ and k is the iteration counter, we arrive at Eq. 4.

Similar to the throughput case, the worst-case latency over the language L is defined as follows.

$$\lambda(L, \boldsymbol{\gamma}_0, \mu) = \max_{\bar{s} \in L} \lambda(\bar{s}, \boldsymbol{\gamma}_0, \mu) \tag{5}$$

In Sections 4 and 5 we provide methods for throughput and latency analysis of FSM-SADF models. The analysis is also applicable to the restricted versions of FSM-SADF such as CSDF and SDF. An SDF graph is an FSM-SADF graph with exactly one scenario. A CSDF graph allows for cyclic variations across different firings of the same actor. This is done by defining a number of *phases* for an actor and deterministic transitions between the phases. A phase defines a fixed execution time and fixed rates for actors. A CSDF actor might have different rates and execution times in different phases. A CSDF can be transformed to an FSM-SADF with a language that contains only one word [12]. The original definition of CSDF does not allow for actors without self-edges; however a CSDF graph that violates this rule can still be transformed to an FSM-SADF and analyzed.

3.5 The Regular Expression Representation

We use regular expressions to formalize a regular language that describes the set of infinite scenario sequences in an FSM-SADF. We use the following syntax for expressions σ.

$$\sigma ::= \rho^\omega \mid \rho\sigma \mid \sigma_1 + \sigma_2$$

The expression ρ^ω formalizes a regular language by an indefinite concatenation of finite words from a non-empty ordinary regular expression (a regular expression that recognizes a language of finite words) ρ. A concatenation $\rho\sigma$ denotes the sequential composition of an ordinary expression ρ and another expression σ. Finally, + denotes the choice between two expressions. We define an ordinary regular expression ρ over a finite alphabet S by the following syntax.

$$\rho ::= \epsilon \mid s \mid \rho_1\rho_2 \mid \rho_1 + \rho_2 \mid \rho^* \mid \rho^n$$

where n is a constant natural number. An ordinary expression ρ can be an empty expression ϵ, a single letter $s \in S$, a sequential composition or a choice between two ordinary expressions, a Kleene iteration $*$ of an expression ρ, and an expression ρ that repeats n times. Although it does not add expressiveness, we add the syntax ρ^n to the common syntax to be able to compactly represent the sequential composition of n times ρ. Note that $(\rho^*)^n$ is not equivalent to $(\rho^n)^*$. $(\rho^*)^n = (\rho^*)(\rho^*)\cdots(\rho^*)$ and $(\rho^n)^* = \epsilon + \rho^n + \rho^{2n} + \cdots$. $\mathcal{L}(\sigma)$ is the language recognized by σ.

4 THROUGHPUT ANALYSIS

To compute the worst-case throughput of an FSM-SADF, we use the state matrices of scenarios and the regular expression representation of the language that specifies the set of possible scenario sequences. We transform the given FSM-SADF to an FSM-SADF with a language that can be formalized by a compact FSM compared to the given FSM. The language defined for the convolution filter in Fig. 2 contains only one sequence, which is expressed by indefinite repetition of an ordinary expression (with in this case a single sequence) $ri^{2W}((ri)^2(cm)^{W-2}(ro)^2)^{H-2}(ro)^{2W}$ that corresponds to production of a full frame. By regarding the frame sequence as a single scenario, called the *frame*

scenario, we transform the given FSM-SADF into an FSM-SADF which has only one scenario and the new expression $frame^{\omega}$ that can be represented by an FSM with only one state.

To complete the transformation, we need to compute the number of outputs and the state matrix of every scenario in the transformed FSM-SADF. The total number of outputs produced during a finite scenario sequence is computed by adding up the outputs produced in every scenario in the sequence. For a sequence $\bar{s} = s_1 \cdots s_n$, the total number of outputs produced during the execution of this sequence is

$$o(\bar{s}) = o(s_1) + \cdots + o(s_n). \tag{6}$$

We compute the state matrix of a scenario sequence by multiplying the state matrices of scenarios in the order that is the reverse of the appearance order of their corresponding scenarios in the sequence. Let $\mathbf{G}(\bar{s})$ denote the state matrix of a sequence $\bar{s} = s_1 \cdots s_n$. Then we have,

$$\mathbf{G}(\bar{s}) = \mathbf{G}_{s_n} \cdots \mathbf{G}_{s_1}. \tag{7}$$

Using a well known method (e.g. [6]) we can transform the expression $frame^{\omega}$ into an ϵ-free FSM and then use the existing analysis of Geilen et al. [11] on the transformed FSM-SADF (in this example it is trivial and it has only one FSM state) to analyze the throughput.

In this particular example we made a new FSM-SADF by defining a single scenario from an ordinary regular expression that formalized a finite sequence of scenarios in the given FSM-SADF. In general, to make a transformation to an FSM-SADF with a compact FSM, we define scenarios for the transformed FSM-SADF from ordinary regular expressions that represent *finite sets* of scenario sequences (instead of a single sequence) in the given FSM-SADF as long as the execution of every sequence in the set produces the same number of outputs. We refer to these expressions as *abstractable* expressions. An abstractable expression does not contain Kleene iterations since they represent infinite sets. Moreover, such an expression does not contain choices between two scenario sequences that produce different numbers of outputs.

For instance let's assume the set of scenario sequences of the example in Fig. 4 is given by $((\phi + \psi\chi^2)\chi^*)^{\omega}$, which is equivalent to the FSM in Fig. 4b. In this expression, $\phi + \psi\chi^2$ is an abstractable sub-expression that can be abstracted into a single scenario in the transformed FSM-SADF. Let's call it scenario υ. This sub-expression formalizes the set that contains the two sequences: ϕ and $\chi\psi^2$. Both of the sequences produce two outputs since $o(\phi) = 2$ and $o(\psi\chi^2) = 0 + 2 \times 1 = 2$; therefore their representation can be abstracted to a single scenario and $o(\upsilon) = 2$. χ^* is the sub-expression that follows $\phi + \psi\chi^2$. Since it contains a Kleene iteration, it can neither be abstracted to a single scenario nor concatenate with $\phi + \psi\chi^2$ and form a larger abstractable sub-expression. Therefore we consider χ as a single scenario in the transformed FSM-SADF. Finally the compact regular expression becomes $(\upsilon\chi^*)^{\omega}$. Recall that the state matrix expresses the worst-case timing relations between initial and final tokens in a scenario. We show that the state matrix of scenario υ is defined as the maximum of the state matrices of the two sequences i.e. $\mathbf{G}_{\upsilon} = \mathbf{G}_{\phi} \oplus \mathbf{G}_{\chi}^2\mathbf{G}_{\psi}$.

We provide a formal definition to transform a given FSM-SADF to a compact FSM-SADF by replacing all abstractable sub-expressions with new scenarios. Observe that the transformation uses a representation of FSM-SADF that defines scenario names, state matrices of scenarios, number of outputs produced in scenarios and a regular expression. Therefore for the transformation, we use a representation of FSM-SADF that includes the mentioned four elements. This representation is defined in the following.

Definition 4.1. Consider an FSM-SADF $A(S, g, r, i, f, o, L)$. Assume $\mathcal{G}$ is a function that maps every graph $g(s)$ to its state matrix $\mathbf{G}_s \in \mathbb{R}_{-\infty}^{f(s)\times i(s)}$. A tuple $M(S, \mathbf{G}, o, \sigma)$ is defined as the matrix-level representation of A, where $\mathbf{G}(s) = \mathcal{G}(g(s))$ maps each scenario to a matrix, σ is such that $L = \mathcal{L}(\sigma)$ and the other two elements preserve their original definitions.

Definition 4.2. Given an FSM-SADF $M(S, \mathbf{G}, o, \sigma)$, the compact transformation of M is an FSM-SADF $M^c(S^c, \mathbf{G}^c, o^c, \sigma^c)$ defined as follows.

The inductively defined function c is used to compute the compacted regular expression $\sigma^c = c(\sigma)$.

$$
\begin{aligned}
c(\sigma):\quad & c(\rho^\omega) = c^\omega(\rho) \\
& c(\rho\sigma) = c(\rho)c(\sigma) \\
& c(\sigma_1 + \sigma_2) = c(\sigma_1) + c(\sigma_2)
\end{aligned}
\qquad
\begin{aligned}
c(\rho):\quad & c(s) = s \\
& c(\rho_1\rho_2) = \begin{cases} s_{\rho_1\rho_2} & \text{if } \rho_1 \text{ and } \rho_2 \text{ are abstractable} \\ c(\rho_1)c(\rho_2) & \text{otherwise} \end{cases} \\
& c(\rho_1 + \rho_2) = \begin{cases} s_{\rho_1+\rho_2} & \text{if } \rho_1 + \rho_2 \text{ is abstractable} \\ c(\rho_1) + c(\rho_2) & \text{otherwise} \end{cases} \\
& c(\rho^n) = \begin{cases} s_{\rho^n} & \text{if } \rho \text{ is abstractable} \\ c^n(\rho) & \text{otherwise} \end{cases} \\
& c(\rho^*) = c^*(\rho)
\end{aligned}
$$

We use s_ρ to name a scenario that abstracts a set of scenario sequences recognized by a sub-expression ρ. Observe that $c(\sigma)$ does not do any compression except on its ordinary sub-expressions ρ, while $c(\rho)$ takes an ordinary regular expression in a recursive fashion and defines new scenarios from abstractable ones. S^c is defined by collecting all scenarios from σ^c. Note that S^c might contain the same scenarios as in S due to $c(s) = s$. For every scenario $s_\rho \in S^c$, the state matrix $\mathbf{G}_{s_\rho} = \mathbf{G}^c(s_\rho)$ represents the worst-case timing relations (between the initial and final tokens) among all sequences recognized by ρ. We compute the functions $\mathbf{G}^c$ and o^c as follows.

$$
\begin{aligned}
& \mathbf{G}^c(s) = \mathbf{G}(s) = \mathbf{G}_s \quad \text{if } s \in S \\
& \mathbf{G}^c(s_{\rho_1\rho_2}) = \mathbf{G}^c(s_{\rho_2})\mathbf{G}^c(s_{\rho_1}) \\
& \mathbf{G}^c(s_{\rho_1+\rho_2}) = \mathbf{G}^c(s_{\rho_1}) \oplus \mathbf{G}^c(s_{\rho_2}) \\
& \mathbf{G}^c(s_{\rho^n}) = (\mathbf{G}^c(s_\rho))^n
\end{aligned}
\qquad
\begin{aligned}
& o^c(s) = o(s) \quad \text{if } s \in S \\
& o^c(s_{\rho_1\rho_2}) = o^c(s_{\rho_1}) + o^c(s_{\rho_2}) \\
& o^c(s_{\rho_1+\rho_2}) = o^c(s_{\rho_1}) = o^c(s_{\rho_2}) \\
& o^c(s_{\rho^n}) = n\, o^c(s_\rho)
\end{aligned}
$$

We need the following two propositions later in the proof of our result.

Proposition 4.3. *Consider an FSM-SADF $M(S, \mathbf{G}, o, \sigma)$ and its compact transformation $M^c(S^c, \mathbf{G}^c, o^c, \sigma^c)$. The following statements hold.*

(1) $\forall s_\rho \in S^c, \mathbf{G}^c(s_\rho) = \bigoplus_{\bar{s} \in \mathcal{L}(\rho)} \mathbf{G}(\bar{s})$. (2) $\forall s_\rho \in S^c$ *and* $\forall \bar{s} \in \mathcal{L}(\rho), o(\bar{s}) = o^c(s_\rho)$.

Proposition 4.4. *Consider an FSM-SADF $M(S, \mathbf{G}, o, \sigma)$ and its compact transformation $M^c(S^c, \mathbf{G}^c, o^c, \sigma^c)$. Let ρ be a sub-expression in σ and ρ^c be the compact transformation of it, i.e. $\rho^c = c(\rho)$. The following statement holds.*

$$\forall \bar{s} \in \mathcal{L}(\rho),\ \exists\, \bar{s}^c \in \mathcal{L}(\rho^c) \text{ s.t. } \mathbf{G}(\bar{s}) \leq \mathbf{G}(\bar{s}^c) \text{ and } o(\bar{s}) = o(\bar{s}^c).$$

The proofs of the above propositions are straightforward using structural induction on ρ. We can now prove the following result.

Theorem 4.5. *An FSM-SADF $M(S, \mathbf{G}, o, \sigma)$ and its compact transformation $M^c(S^c, \mathbf{G}^c, o^c, \sigma^c)$ have the same throughput.*

Proof. We divide the proof into two parts, first proving that for every scenario sequence $\bar{s} \in \mathcal{L}(\sigma)$ there exist a scenario sequence $\bar{s}^c \in \mathcal{L}(\sigma^c)$ such that $\tau(\bar{s}^c) \leq \tau(\bar{s})$ and then proving that for every $\bar{s}^c$ there exists an $\bar{s}$ such that $\tau(\bar{s}) \leq \tau(\bar{s}^c)$. We prove the first part by structural induction on σ.

- For the base case i.e. in case $\sigma = \rho^\omega$, let $\bar{s} = \bar{t}_1\bar{t}_2\bar{t}_3\cdots$ such that $\bar{t}_i \in \mathscr{L}(\rho)$. According to Proposition 4.4, for any $\bar{t}_i$, there exists a sequence $\bar{t}_i^c$ such that $\mathbf{G}(\bar{t}_i) \leq \mathbf{G}(\bar{t}_i^c)$ and $o(\bar{t}_i) = o(\bar{t}_i^c)$. We let $\bar{s}^c = \bar{t}_1^c\bar{t}_2^c\bar{t}_3^c\cdots$. To show that $\tau(\bar{s}^c) \leq \tau(\bar{s})$ let $\bar{s}_i = \bar{t}_1\bar{t}_2\cdots\bar{t}_i$ and $\bar{s}_i^c = \bar{t}_1^c\bar{t}_2^c\cdots\bar{t}_i^c$. We know that for any i, $\mathbf{G}(\bar{s}_i) \leq \mathbf{G}(\bar{s}_i^c)$ and $o(\bar{s}_i) = o(\bar{s}_i^c)$. Hence we have

$$\tau(\bar{s}) = \lim_{i\to\infty}\sup\frac{\Sigma_{n=1}^{i} o(\bar{s}_i)}{\left\|\mathbf{G}(\bar{s}_i)\boldsymbol{\gamma}_0\right\|} \geq \lim_{i\to\infty}\sup\frac{\Sigma_{n=1}^{i} o(\bar{s}_i^c)}{\left\|\mathbf{G}(\bar{s}_i^c)\boldsymbol{\gamma}_0\right\|} = \tau(\bar{s}^c).$$

- In case $\sigma = \rho\sigma_2$, let $\bar{s} = \bar{s}_1\bar{s}_2$ such that $\bar{s}_1 = \mathscr{L}(\rho)$ and $\bar{s}_2 \in \mathscr{L}(\sigma_2)$. Note that the throughput of $\bar{s}$ is equal to the throughput of $\bar{s}_2$, since $\bar{s}_1$ is only a finite prefix of $\bar{s}$ and it does not affect the throughput. According to the induction hypothesis there exists a sequence $\bar{s}_2^c$ such that $\tau(\bar{s}_2) \geq \tau(\bar{s}_2^c)$. Hence we have $\tau(\bar{s}) = \tau(\bar{s}_2) \geq \tau(\bar{s}_2^c)$.
- In case $\sigma = \sigma_1 + \sigma_2$. Consider σ_1^c and σ_2^c such that $c(\sigma_1 + \sigma_2) = c(\sigma_1) + c(\sigma_2) = \sigma_1^c + \sigma_2^c$. Without loss of generality, let $\bar{s}$ belong to $\mathscr{L}(\sigma_1)$. According to the induction hypothesis there exists $\bar{s}_1^c \in \mathscr{L}(\sigma_1^c)$ such that $\tau(\bar{s}) \geq \tau(\bar{s}_1^c)$.

For the second part of the proof let $\bar{s}^c = s_1^c s_2^c s_3^c\cdots$ where $s_k^c \in S^c$. According to [9], for any scenario sequence $\bar{s}^c$ there exists a periodic scenario sequence $\bar{s}_o^c = (s_1^c s_2^c\cdots s_n^c)^\omega$ for some n, such that $\tau(\bar{s}_o^c) \leq \tau(\bar{s}^c)$. Therefore to prove the second part, it suffices to show that there exist a sequence $\bar{s}$ such that $\tau(\bar{s}) \leq \tau(\bar{s}_o^c)$. The throughput of a periodic sequence is limited by the critical cycle in a graph ((max, +) automaton) that encodes the dependencies between all initial and final tokens in the sequence [11]. The nodes in this graph represent the initial/final tokens in each of the scenarios. The state matrices are used to connect the nodes to each other. For every non $-\infty$ element in the matrix there exists an edge between the corresponding initial/final tokens, labelled with the value of that element and with the number of outputs produced in that scenario. The critical cycle is composed of critical edges and has the lowest output/time ratio.

Now let $\mathbf{G}_{ij}$ denote the element on row i, column j of matrix $\mathbf{G}$. The critical cycle of the graph for $\bar{s}_o^c = (s_1^c s_2^c\cdots s_n^c)^\omega$ specifies a pair $((i,j)_k, o_k)$ for each scenario s_k^c, such that $\mathbf{G}_{ij}(s_k^c)$ is the critical element (the element corresponding to the critical edge) in the state matrix of s_k^c, and $o_k = o(s_k^c)$. Now let ρ_k be the sub-expression abstracted by s_k^c. It follows from Proposition 4.3 that $\bar{s}_k = \text{argmax}_{\bar{t}_k\in\mathscr{L}(\rho_k)}\,\mathbf{G}_{ij}(\bar{t}_k)$ exists such that $\mathbf{G}_{ij}(\bar{s}_k) = \mathbf{G}_{ij}(s_k^c)$ and $o(\bar{s}_k) = o_k$. Now if we let $\bar{s} = (\bar{s}_1\bar{s}_2\cdots\bar{s}_n)^\omega$, there exists a cycle in the graph of $\bar{s}$ with the same cycle ratio as $\bar{s}_o^c$. Therefore the graph of $\bar{s}$ has a critical cycle which has at most the same ratio as $\bar{s}^c$, i.e. $\tau(\bar{s}) \leq \tau(\bar{s}_o^c)$. Note that if the periodic sub-sequence $\bar{s}_o^c$ is in the language of a sub-expression in σ^c, then $\bar{s}$ belongs to the language of a sub-expression in σ. □

We provide an algorithm for the throughput analysis of FSM-SADF graphs. The algorithm first performs the FSM-SADF transformation defined by Def. 4.2, and then uses a conversion from regular expressions to an ϵ-free FSMs for the final analysis on the transformed FSM-SADF. Next we discuss the complexity of this algorithm. We assume a regular expression σ is given by some syntax tree t_σ. The leaves of the syntax tree are labelled by scenarios $s \in S$, and the inner nodes are either a binary operator labelled + or $\cdot$ (sequential composition), or they are a unary operator and labelled by $*$, ω or a natural number n (for the constant repetition). Computing $c(\sigma)$, $\mathbf{G}^c$ and o^c can be done by a reversed preorder traversal of t_σ. The time complexity of computing $c(\sigma)$ and o^c is linear in the size (the number of nodes) of t_σ. For a + or $\cdot$ node, computing $\mathbf{G}^c$ has a constant complexity and for an n node it has a logarithmic complexity in the value of n. It is common to use Polish Notation (PN) to represent σ. If we represent the value n with $\log(n)$ digits and use a character to represent each scenario, then the complexity of the transformation by Def. 4.2 is linear in the length of the representation of σ.

For deterministic cases of FSM-SADF i.e. SDF and CSDF, the transformed FSM-SADF will always have a regular expression that represents indefinite repetition of a single scenario. Such an expression can be transformed to an ϵ-free FSM with only one state, in constant time. Therefore the throughput computation for deterministic cases of FSM-SADF is always linear in the length of the regular expression. For non-deterministic cases of FSM-SADF, if for all sub-expressions ρ^n in σ, ρ are abstractable, then the transformed FSM-SADF will have a conventional regular expression (without a repetition construct). A sequential algorithm is provided by Hagenah et al. for conversion from a conventional regular expression to an ϵ-free FSM that takes $\mathcal{O}(k \log^2(k))$ time, where k is the length of the regular expression [17]. If in the worst-case, the length of the expression for the transformed FSM-SADF is equal to the length of the expression in the given FSM-SADF, then the throughput computation will have a polynomial complexity. For sub-expressions ρ^n that are not abstractable, the conversion to FSM is pseudo-polynomial in n, which makes the throughput computations pseudo-polynomial in the length of the regular expression.

5 LATENCY ANALYSIS

This section provides a compositional latency analysis for an FSM-SADF, of which the language is given by a regular expression. We first provide a latency analysis for a set of finite SADF scenario sequences given by an ordinary regular expression, as a means to compute the latency of the set of infinite scenario sequences of an FSM-SADF. A naive method to compute the latency of a finite scenario sequence is to compute the production times of all outputs and use Eq. 4 where k has an upper bound. Consider again the scenario sequence for producing a 9×11 frame in the filter example i.e. $(ri)^{18}((ri)^2(cm)^7(ro)^2)^9(ro)^{18}$. Scenario cm is the first scenario that appears in the sequence that produces output. This scenario follows the scenario sequence that corresponds to the frame rush-in and the line rush-in phases. To proceed, we provide the output matrices: $\mathbf{H}_{cm} = \begin{bmatrix} 13 & 11 & 2 \end{bmatrix}$ and $\mathbf{H}_{ro} = \begin{bmatrix} -\infty & 9 & 2 \end{bmatrix}$. Assuming the initial state vector $\mathbf{0}$, the outputs are produced at

$$p_0 = \mathbf{H}_{cm}\mathbf{G}_{ri}^{20}\mathbf{0} = 53 \quad p_1 = \mathbf{H}_{cm}\mathbf{G}_{cm}\mathbf{G}_{ri}^{20}\mathbf{0} = 57 \qquad \cdots \qquad p_{98} = \mathbf{H}_{ro}\mathbf{G}_{ro}^{17}\left(\mathbf{G}_{ro}^{2}\mathbf{G}_{cm}^{7}\mathbf{G}_{ri}^{2}\right)^{9}\mathbf{G}_{ri}^{18}\mathbf{0} = 440.$$

Using Eq. 4, the latency of producing pixels in one frame relative to period 5 (derived from the earlier computed max throughput of 99/432) is 53. According to Fig. 3 the first output contributes to the latency. In the beginning of the execution the convolution filter goes through the frame and line rush-in phases where there are no outputs produced. However, after getting past theses phases, outputs are produced more frequently. Assuming that the first start time of the external actor (see the definition of latency in Section 3.4) is 53, the next outputs will be available before the next starts of the external actor with $\mu = 5$ (this is true only when the first output causes the highest latency).

Although computing the production times of all outputs one by one is a solution to determine the latency of a sequence, it would scale only linearly in the total number of scenarios in the sequence. Therefore, we provide a compositional method to efficiently compute the latency of a finite SADF scenario sequence. In this section $\bar{s}$ denotes a finite sequence unless stated otherwise. For readability, we leave μ implicit and write $\lambda(\bar{s}, \boldsymbol{\gamma}_0)$ instead of $\lambda(\bar{s}, \boldsymbol{\gamma}_0, \mu)$. For all examples in this section we assume $\mu = 5$ and $\boldsymbol{\gamma}_0 = \mathbf{0}$.

A scenario sequence is either a single scenario or the sequential composition of two sub-sequences. In case $\bar{s}$ is a single scenario $\bar{s} = s$ that produces $o(s) > 0$ outputs during its execution, we can use the output matrix to capture the output production times and use them to compute the latency. For instance, for scenario cm we can obtain the latency as follows.

$$\lambda(cm, \boldsymbol{\gamma}_0) = p_0 - 5 \times 0 = \mathbf{H}_{cm}\mathbf{0} - 0 = \begin{bmatrix} 13 & 11 & 2 \end{bmatrix}\mathbf{0} = 13$$

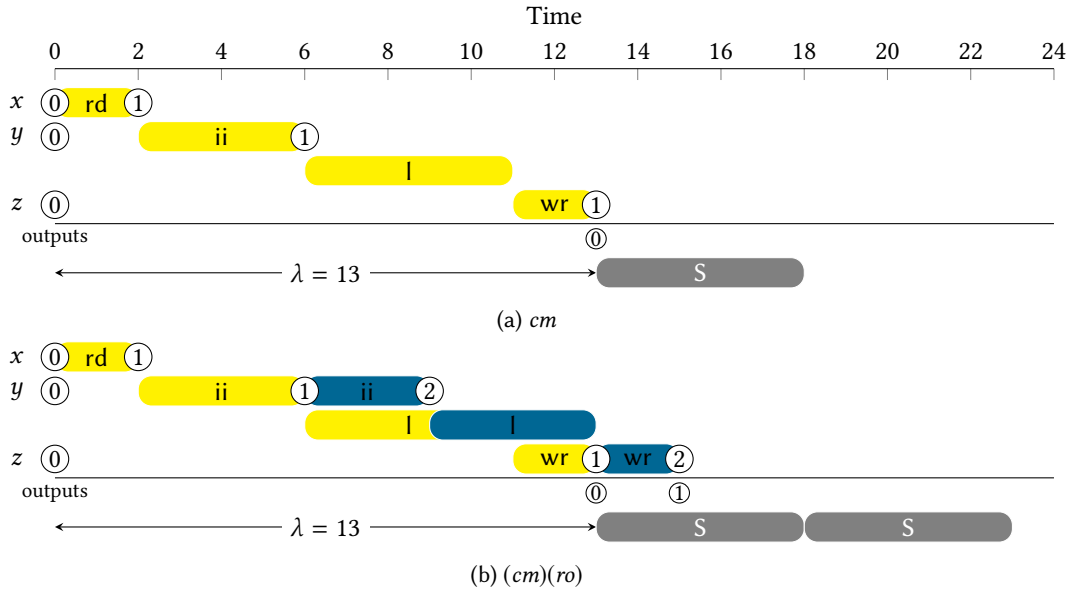


Fig. 6. Execution traces for two example scenario sequences.

In general, for a scenario s, the k^{th} element of $\mathbf{H}_s\boldsymbol{\gamma}_0$ corresponds to the production time of the k^{th} output in the scenario. Therefore, we can obtain the latency as follows.

$$\lambda(s,\boldsymbol{\gamma}_0) = \max_{0\le k<o(s)} p_k - \mu k = \max_{0\le k<o(s)} (\mathbf{H}_s\boldsymbol{\gamma}_0)_k - \mu k$$

Alternatively, we can define the *compensation matrix*,

$$\mathbf{M}(s) = \begin{bmatrix} 0 & -\mu & -2\mu & \cdots & (o(s)-1)\mu \end{bmatrix},$$

that captures the number of periods occurring till the corresponding output. Then we compute the latency of s in matrix form as

$$\lambda(s,\boldsymbol{\gamma}_0) = \mathbf{M}(s)\mathbf{H}_s\boldsymbol{\gamma}_0 = \left[\mathbf{H}_s^T\mathbf{M}^T(s)\right]^T\boldsymbol{\gamma}. \tag{8}$$

Observe that Eq. 8 computes the following vector.

$$\boldsymbol{\lambda}(s) = \mathbf{H}_s^T\mathbf{M}^T(s). \tag{9}$$

We refer to $\boldsymbol{\lambda}(s)$ as the *latency vector* of scenario s. The latency vector $\boldsymbol{\lambda}(s)$ is a column vector that expresses the timing relations between the availability times of labelled initial tokens and the production times of outputs relative to the required period μ, during the execution of the scenario. According to Eq. 8 the latency equals the inner product of the latency vector and the initial state vector, i.e.,

$$\lambda(s,\boldsymbol{\gamma}_0) = \boldsymbol{\lambda}^T(s)\boldsymbol{\gamma}_0. \tag{10}$$

Let's consider an example of this case in which $s = cm$. Using the above equation, the latency is computed as follows.

$$\boldsymbol{\lambda}(cm) = \begin{bmatrix} 13 & 11 & 2 \end{bmatrix}^T \begin{bmatrix} 0 \end{bmatrix} = \begin{bmatrix} 13 & 11 & 2 \end{bmatrix}^T$$
$$\lambda(cm,\mathbf{0}) = \begin{bmatrix} 13 & 11 & 2 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T = 13$$

Fig. 6a shows the execution trace of scenario *cm* followed by the execution of the external actor S that needs this output. The output is produced at 13. Therefore the latency is 13 since the start of actor S cannot happen earlier.

Now let $\bar{s}$ be the sequential composition of two scenario sequences i.e. $\bar{s} = \bar{s}_1\bar{s}_2$. We work out an example where $\bar{s}_1 = cm$ and $\bar{s}_2 = ro$ i.e. $\bar{s} = (cm)(ro)$. We need to compute the maximum of the latencies of both scenarios. The latency of *cm* is 13 (using Eq. 8). Since *ro* follows *cm*, the initial state vector for *ro* is $\boldsymbol{\gamma}_1 = \mathbf{G}_{cm}\boldsymbol{\gamma}_0 = \begin{bmatrix} 2 & 6 & 13 \end{bmatrix}^T$. This can be observed in Fig. 6b by circles marked with number one in the x, y, z rows. To compute the latency of *ro* as the second scenario in the sequence $(cm)(ro)$, we also need to account for the output produced in *cm*, because the first firing of the external actor S consumes this output. Therefore, we need to compute the latency of *ro* with the state vector $\boldsymbol{\gamma}_1$ and subtract 5 from the result to compensate for the output produced by *cm*. Using Eq. 8 with $\boldsymbol{\gamma}_1$ for *ro* returns 15 and, subtracting 5 from 15 we obtain the latency of 10 for *ro*. The latency of the entire sequence is the maximum of 13 and 10, which is 13.

In general, in a sequence s_1s_2, to compute the latency of the second scenario s_2, we need to compute the latency of s_2 with the state vector $\boldsymbol{\gamma}_1$ and subtract $o(s_1)\mu$ from the result to compensate for all outputs produced in s_1, i.e. we compute $\lambda(s_2, \boldsymbol{\gamma}_1) - o(s_1)\mu$. Alternatively, we can consider the compensation term $-o(s_1)\mu$ in computing the latency of the second scenario by normalizing the state vector $\boldsymbol{\gamma}_1$ as follows.

$$\hat{\boldsymbol{\gamma}}_1 = \boldsymbol{\gamma}_1 - o(s_1)\mu = \mathbf{G}_{s_1}\boldsymbol{\gamma}_0 - o(s_1)\mu = (\mathbf{G}_{s_1} - o(s_1)\mu)\boldsymbol{\gamma}_0.$$

We normalized the state vector by multiplying the initial state vector by the following matrix.

$$\hat{\mathbf{G}}(s) = \mathbf{G}_s - o(s)\mu. \tag{11}$$

We define the *normalized state matrix* $\hat{\mathbf{G}}(s)$ as the state matrix of scenario s that is normalized for its outputs. Now the latency over the sequence s_1s_2 can be obtained as follows.

$$\lambda(s_1s_2, \boldsymbol{\gamma}_0) = \lambda(s_1, \boldsymbol{\gamma}_0) \oplus \lambda(s_2, \hat{\mathbf{G}}(s_1)\boldsymbol{\gamma}_0) = \boldsymbol{\lambda}^T(s_1)\boldsymbol{\gamma}_0 \oplus \boldsymbol{\lambda}^T(s_2)\hat{\mathbf{G}}(s_1)\boldsymbol{\gamma}_0 = \left[\boldsymbol{\lambda}^T(s_1) \oplus \boldsymbol{\lambda}^T(s_2)\hat{\mathbf{G}}(s_1)\right]\boldsymbol{\gamma}_0 \tag{12}$$

Hence the latency vector for $\bar{s} = s_1s_2$ is

$$\boldsymbol{\lambda}(s_1s_2) = \left[\boldsymbol{\lambda}^T(s_1) \oplus \boldsymbol{\lambda}^T(s_2)\hat{\mathbf{G}}(s_1)\right]^T = \boldsymbol{\lambda}(s_1) \oplus \hat{\mathbf{G}}^T(s_1)\boldsymbol{\lambda}(s_2) \tag{13}$$

Again considering the sequence $(cm)(ro)$, we compute the normalized state matrix of *cm* (Eq. 11) and the latency vector of *ro* (Eq. 9) as follows.

$$\hat{\mathbf{G}}(cm) = \mathbf{G}_{cm} - o(cm) \times \mu = \begin{bmatrix} 2 & -\infty & -\infty \\ 6 & 4 & -\infty \\ 13 & 11 & 2 \end{bmatrix} - 1 \times 5 = \begin{bmatrix} -3 & -\infty & -\infty \\ 1 & -1 & -\infty \\ 8 & 6 & -3 \end{bmatrix}$$

$$\boldsymbol{\lambda}(ro) = \begin{bmatrix} -\infty & 9 & 2 \end{bmatrix}^T \begin{bmatrix} 0 \end{bmatrix} = \begin{bmatrix} -\infty & 9 & 2 \end{bmatrix}^T$$

The latency according to Eq. 12 is computed as follows.

$$\lambda((cm)(ro), \boldsymbol{\gamma}_0) = \left[\begin{bmatrix} 13 & 11 & 2 \end{bmatrix} \oplus \begin{bmatrix} -\infty & 9 & 2 \end{bmatrix} \begin{bmatrix} -3 & -\infty & -\infty \\ 1 & -1 & -\infty \\ 8 & 6 & -3 \end{bmatrix}\right] \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = 13$$

From Fig. 6b, we can obtain the output production times and compute the latency according to Eq. 4 as $\lambda((cm)(ro), \mathbf{0}) = (13 - 0) \oplus (15 - 5) = 13$. This confirms the result computed earlier from Eq. 12.

Note that using Eq. 12, the latency of a sequence composed of two scenarios is obtained by first computing a vector that is then multiplied by the initial state vector. This vector extends the concept

of the latency vector of a scenario to a sequence. In general, for any finite sequence $\bar{s} = s_1 \cdots s_n$ the latency vector $\boldsymbol{\lambda}(\bar{s})$ is defined as follows.

$$\boldsymbol{\lambda}(\bar{s}) = \bigoplus_{0<k\leq n} \left(\hat{\mathbf{G}}^T(s_1 \cdots s_{k-1})\boldsymbol{\lambda}(s_k)\right) \tag{14}$$

In Eq. 14, the normalized state matrix $\hat{\mathbf{G}}(\bar{s})$ of a sequence $\bar{s}$ is the state matrix of the sequence that accounts for all outputs produced in the sequence i.e.

$$\hat{\mathbf{G}}(\bar{s}) = \mathbf{G}(\bar{s}) - o(\bar{s}). \tag{15}$$

In Eq. 14, for $k = 1$, $s_1 \cdots s_{k-1}$ represents the empty scenario sequence ϵ and $\hat{\mathbf{G}}(\epsilon)$ is defined to be the identity (max, +) matrix $\mathcal{I}$. We show that the latency of a sequence equals the inner product of the latency vector of the sequence and the initial state vector.

Proposition 5.1. *The latency of any finite scenario sequence $\bar{s} = s_1 \cdots s_n$ is equal to the inner product of its latency vector and the initial state vector i.e.*

$$\lambda(\bar{s}, \boldsymbol{\gamma}_0) = \boldsymbol{\lambda}^T(\bar{s})\boldsymbol{\gamma}_0. \tag{16}$$

Proof. First we distribute the max term in the definition of the latency (Eq.4) into n max terms each of which corresponds to the latency of outputs produced during a single scenario in the sequence.

$$\begin{aligned}
\lambda(\bar{s}, \boldsymbol{\gamma}_0) &= \max_{0\leq k<o(\bar{s})}(p_k - k\mu) \\
&= \max_{0\leq k<o(s_1)}(p_k - k\mu) \oplus \max_{o(s_1)\leq k<o(s_1 s_2)}(p_k - k\mu) \oplus \cdots \oplus \max_{o(s_1\cdots s_{n-1})\leq k<o(\bar{s})}(p_k - k\mu) \\
&= \max_{0\leq k<o(s_1)}(p_k - k\mu) \oplus \max_{0\leq k<o(s_2)}(p_{k+o(s_1)} - k\mu) - o(s_1)\mu \oplus \cdots \\
&\quad \oplus \max_{0\leq k<o(s_n)}(p_{k+o(s_1\cdots s_{n-1})} - k\mu) - o(s_1 \cdots s_{n-1})\mu.
\end{aligned} \tag{17}$$

Recall that the production time of outputs in scenario s_i within a scenario sequence, can be computed in vector $\mathbf{H}_{s_i}\mathbf{G}(s_1 \cdots s_{i-1})\boldsymbol{\gamma}_0$. Using the compensation matrix $\mathbf{M}(s)$ we can rewrite each of the max terms in the following form.

$$\max_{0\leq k<o(s_i)}(p_{k+o(s_1\cdots s_{i-1})} - k\mu) - o(s_1 \cdots s_{i-1})\mu$$
$$= \begin{bmatrix} p_{o(s_1\cdots s_{i-1})} - 0 & p_{o(s_1\cdots s_{i-1})+1} - \mu & p_{o(s_1\cdots s_{i-1})+2} - 2\mu & \cdots & p_{o(s_1\cdots s_{i-1})+o(s_i)-1} - (o(s_i)-1)\mu \end{bmatrix}$$
$$- o(s_1 \cdots s_{i-1})\mu = \mathbf{M}(s_i)\mathbf{H}_{s_i}\mathbf{G}(s_1 \cdots s_{i-1})\boldsymbol{\gamma}_0 - o(s_1 \cdots s_{i-1})\mu = \mathbf{M}(s_i)\mathbf{H}_{s_i}\hat{\mathbf{G}}(s_1 \cdots s_{i-1})\boldsymbol{\gamma}_0.$$

By substituting the max terms in Eq.17 with their matrix forms above, we get

$$\lambda(\bar{s}, \boldsymbol{\gamma}_0) = \left[\bigoplus_{0<k\leq n}\left(\mathbf{M}(s_k)\mathbf{H}_{s_k}\hat{\mathbf{G}}(s_1 \cdots s_{k-1})\right)\right]\boldsymbol{\gamma}_0 = \left[\bigoplus_{0<k\leq n}\left(\boldsymbol{\lambda}^T(s_k)\hat{\mathbf{G}}(s_1 \cdots s_{k-1})\right)\right]\boldsymbol{\gamma}_0 = \boldsymbol{\lambda}^T(\bar{s})\boldsymbol{\gamma}_0.$$

□

Now we show how to compositionally compute the latency vector of a sequence by splitting the sequence from any arbitrary point in the sequence. We show that the latency vector of $\bar{s} = \bar{s}_1\bar{s}_2$ can be computed from the latency vectors of $\bar{s}_1$ and $\bar{s}_2$ similar to the sequential composition of two scenarios (Eq. 13) as follows.

$$\boldsymbol{\lambda}(\bar{s}_1\bar{s}_2) = \boldsymbol{\lambda}(\bar{s}_1) \oplus \hat{\mathbf{G}}^T(\bar{s}_1)\boldsymbol{\lambda}(\bar{s}_2). \tag{18}$$

Proposition 5.2. *Consider a finite sequence $\bar{s} = s_1 \cdots s_m s_{m+1} \cdots s_n$. Let's assume $\bar{s}_1 = s_1 \cdots s_m$ and $\bar{s}_2 = s_{m+1} \cdots s_n$. The latency vector $\boldsymbol{\lambda}(\bar{s})$ can be computed by Eq.18.*

Proof.

$$\begin{aligned}
\boldsymbol{\lambda}^T(\bar{s}) =& \boldsymbol{\lambda}^T(s_1 \cdots s_m s_{m+1} \cdots s_n) = \left[\bigoplus_{0<k\leq m} \boldsymbol{\lambda}^T(s_k)\hat{\mathbf{G}}(s_1 \cdots s_{k-1})\right] \oplus \left[\bigoplus_{m<k\leq n} \boldsymbol{\lambda}^T(s_k)\hat{\mathbf{G}}(s_1 \cdots s_{k-1})\right] \\
=& \boldsymbol{\lambda}^T(\bar{s}_1) \oplus \left[\bigoplus_{m<k\leq n} \boldsymbol{\lambda}^T(s_k)\mathbf{G}(s_1 \cdots s_{k-1}) - o(s_1 \cdots s_{k-1})\right] \\
=& \boldsymbol{\lambda}^T(\bar{s}_1) \oplus \left[\bigoplus_{m<k\leq n} \boldsymbol{\lambda}^T(s_k)\mathbf{G}(s_{m+1} \cdots s_{k-1})\mathbf{G}(s_1 \cdots s_m) - o(s_{m+1} \cdots s_{k-1}) - o(s_1 \cdots s_m)\right] \\
=& \boldsymbol{\lambda}^T(\bar{s}_1) \oplus \left[\bigoplus_{m<k\leq n} \boldsymbol{\lambda}^T(s_k)\left[\mathbf{G}(s_{m+1} \cdots s_{k-1}) - o(s_{m+1} \cdots s_{k-1})\right]\left[\mathbf{G}(s_1 \cdots s_m) - o(s_1 \cdots s_m)\right]\right] \\
=& \boldsymbol{\lambda}^T(\bar{s}_1) \oplus \left[\bigoplus_{m<k\leq n} \boldsymbol{\lambda}^T(s_k)\mathbf{G}(s_{m+1} \cdots s_{k-1}) - o(s_{m+1} \cdots s_{k-1})\right]\left[\mathbf{G}(s_1 \cdots s_m) - o(s_1 \cdots s_m)\right] \\
=& \boldsymbol{\lambda}^T(\bar{s}_1) \oplus \left[\bigoplus_{m<k\leq n} \boldsymbol{\lambda}^T(s_k)\hat{\mathbf{G}}(s_m \cdots s_{k-1})\right]\hat{\mathbf{G}}(\bar{s}_1) = \boldsymbol{\lambda}^T(\bar{s}_1) \oplus \boldsymbol{\lambda}^T(\bar{s}_2)\hat{\mathbf{G}}(\bar{s}_1)
\end{aligned}$$

By taking the transpose of both sides of the above equation we obtain Eq.18. □

Next we show that we can use Eq. 18 to efficiently compute the latency of repetitive sequences. To compute the latency vector of a finite sequence that repeats n times i.e. $\bar{s}^n$, we can recursively use Eq. 18 as follows.

$$\begin{aligned}
\boldsymbol{\lambda}(\bar{s}^n) = \boldsymbol{\lambda}(\bar{s}\bar{s}^{n-1}) = \boldsymbol{\lambda}(\bar{s}) \oplus \hat{\mathbf{G}}^T(\bar{s})\boldsymbol{\lambda}(\bar{s}^{n-1}) = \boldsymbol{\lambda}(\bar{s}) \oplus \hat{\mathbf{G}}^T(\bar{s})\left(\boldsymbol{\lambda}(\bar{s}) \oplus \hat{\mathbf{G}}^T(\bar{s})\boldsymbol{\lambda}(\bar{s}^{n-2})\right) \\
= \left(\bigoplus_{0\leq k<n} (\hat{\mathbf{G}}^T(\bar{s}))^k\right)\boldsymbol{\lambda}(\bar{s}) \qquad (19)
\end{aligned}$$

Computation of Eq. 19 looks linear in n; however, it can be computed with the worst-case complexity of $\mathcal{O}(\log n)$. Let's assume $\mathcal{A}(V)$ is the adjacency graph of $\mathbf{G}$ where V is the number of vertices of $\mathcal{A}$. When $n > V$, and there are no positive cycles in $\mathcal{A}$, computing $\bigoplus_{0\leq k<n} \mathbf{G}^k$ is to compute the longest paths from all nodes to all nodes in $\mathcal{A}$, which has worst-case complexity of $\mathcal{O}(V^3)$ using the Floyd-Warshall algorithm [7]. When $n > V$, and there is a positive cycle in $\mathcal{A}$, $\bigoplus_{0\leq k<n} \mathbf{G}^k$ can be computed with the complexity of $\mathcal{O}(\log n)$, because $\bigoplus_{0\leq k<n} \mathbf{G}^k = (\mathcal{I} \oplus \mathbf{G})^{n-1}$ and rasing a matrix to a power can be computed with log complexity. When $n \leq V$, the complexity of computing $\bigoplus_{0\leq k<n} \mathbf{G}^k$ is linear in n. For multi-scale dataflow models, typically $n \gg V$ since, n indicates the number of repetitions of an execution and V indicates the number of initial tokens in the dataflow model. In the convolution filter example, the scenario repetitions can be as large as the number of lines or columns of the input frame e.g. 1000 but the number of initial tokens is just 3.

We introduce an approach to compute the latency of a set of sequences based on the compositional latency analysis provided for a single sequence. Since we are dealing with a set of sequences, we first generalize the latency vector concept defined for a single sequence to a set of sequences. To obtain the worst-case latency of the set, we define the latency vector $\boldsymbol{\lambda}(\rho)$ of an ordinary regular expression ρ as the maximum latency vector over all sequences in the language of ρ:

$$\boldsymbol{\lambda}(\rho) = \bigoplus_{\bar{s}\in\mathcal{L}(\rho)} \boldsymbol{\lambda}(\bar{s}). \qquad (20)$$

The goal is to do compositional latency analysis for a language of scenario sequences similar to what we did for a single sequence case. Recall that to compute the latency of a sequence $\bar{s}_2$ that follows sequence $\bar{s}_1$, we had to compute a normalized state matrix that accounts for the outputs produced during $\bar{s}_1$. In the case of expressions, to compute the latency of ρ_2 that follows ρ_1, we need to obtain the worst-case normalized state matrix over all sequences recognized by ρ_1. Therefore we define the normalized matrix $\hat{\mathbf{G}}(\rho)$ of a regular expression ρ as the maximum of normalized matrices over all sequences in the language of ρ i.e.

$$\hat{\mathbf{G}}(\rho) = \bigoplus_{\bar{s}\in\mathscr{L}(\rho)} \hat{\mathbf{G}}(\bar{s}). \tag{21}$$

Note that we can not use Eq.21 to compute $\hat{\mathbf{G}}(\rho)$ since $\mathscr{L}(\rho)$ might be an infinite set. Therefore we provide a recursive method to compute it and later we prove that the computation corresponds to the definition. The computation is defined as follows.

$$\hat{\mathbf{G}}(\rho): \quad \hat{\mathbf{G}}(\rho_1\rho_2) = \hat{\mathbf{G}}(\rho_2)\hat{\mathbf{G}}(\rho_1) \tag{22}$$

$$\hat{\mathbf{G}}(\rho_1 + \rho_2) = \hat{\mathbf{G}}(\rho_1) \oplus \hat{\mathbf{G}}(\rho_2) \tag{23}$$

$$\hat{\mathbf{G}}(\rho^*) = \hat{\mathbf{G}}^*(\rho) \tag{24}$$

$$\hat{\mathbf{G}}(\rho^n) = \hat{\mathbf{G}}^n(\rho) \tag{25}$$

Now we provide a method to efficiently compute the latency vector of an ordinary regular expression ρ for every syntactic composition in an ordinary regular expression as follows.

$$\boldsymbol{\lambda}(\rho): \quad \boldsymbol{\lambda}(\rho_1\rho_2) = \boldsymbol{\lambda}(\rho_1) \oplus \hat{\mathbf{G}}^T(\rho_1)\boldsymbol{\lambda}(\rho_2) \tag{26}$$

$$\boldsymbol{\lambda}(\rho_1 + \rho_2) = \boldsymbol{\lambda}(\rho_1) \oplus \boldsymbol{\lambda}(\rho_2) \tag{27}$$

$$\boldsymbol{\lambda}(\rho^*) = (\hat{\mathbf{G}}^*(\rho))^T\boldsymbol{\lambda}(\rho) \tag{28}$$

$$\boldsymbol{\lambda}(\rho^n) = \bigoplus_{0\le k<n} (\hat{\mathbf{G}}^T(\rho))^k\boldsymbol{\lambda}(\rho) \tag{29}$$

Note that if for an expression ρ the star closure $\hat{\mathbf{G}}^*(\rho)$ in Eq. 28 or Eq. 24 does not exist, it means that the production of outputs during at least one sequence in the language of that expression cannot keep up with the consumption of the outputs with period μ i.e. the throughput is too low and therefore the latency is not defined.

Now we provide proof of correctness for latency vector computations for ordinary regular expressions. We first show that using Eqs. 22-25 we can compute the normalized matrix of any recursive combination of ordinary regular expressions.

Proposition 5.3. *For any ordinary expression ρ, $\hat{\mathbf{G}}(\rho)$ computed by Eqs. 22-25 satisfies $\hat{\mathbf{G}}(\rho) = \bigoplus_{\bar{s}\in\mathscr{L}(\rho)} \hat{\mathbf{G}}(\bar{s})$.*

Proof. We prove this by structural induction on ρ. The base case $\rho = s$ is trivial. For the induction steps we have:

$$\hat{\mathbf{G}}(\rho_1\rho_2) = \bigoplus_{\bar{s}_1\bar{s}_2\in\mathscr{L}(\rho_1\rho_2)} \mathbf{G}(\bar{s}_1\bar{s}_2) - o(\bar{s}_1\bar{s}_2) = \bigoplus_{\bar{s}_1\in\mathscr{L}(\rho_1),\bar{s}_2\in\mathscr{L}(\rho_2)} \mathbf{G}(\bar{s}_2)\mathbf{G}(\bar{s}_1) - o(\bar{s}_1) - o(\bar{s}_2)$$
$$= \bigoplus_{\bar{s}_2\in\mathscr{L}(\rho_2)} \mathbf{G}(\bar{s}_2) - o(\bar{s}_2) \bigoplus_{\bar{s}_1\in\mathscr{L}(\rho_1)} \mathbf{G}(\bar{s}_1) - o(\bar{s}_1) = \hat{\mathbf{G}}(\rho_2)\hat{\mathbf{G}}(\rho_1).$$

$$\hat{\mathbf{G}}(\rho^*) = \bigoplus_{\bar{s}\in\bigcup_{n\ge0}\mathscr{L}(\rho^n)} \hat{\mathbf{G}}(\bar{s}) = \bigoplus_{n\ge0} \bigoplus_{\bar{s}\in\mathscr{L}(\rho^n)} \hat{\mathbf{G}}(\bar{s}) = \mathcal{I} \oplus \hat{\mathbf{G}}(\rho) \oplus \hat{\mathbf{G}}^2(\rho) \oplus \cdots = \hat{\mathbf{G}}^*(\rho).$$

The proof for $\hat{\mathbf{G}}(\rho^n)$ follows straightforwardly from $\hat{\mathbf{G}}(\rho_1\rho_2)$, and for $\hat{\mathbf{G}}(\rho_1 + \rho_2)$ it is trivial. □

Proposition 5.4. *For any ordinary expression ρ, $\boldsymbol{\lambda}(\rho)$ computed by Eqs. 26-29 satisfies $\boldsymbol{\lambda}(\rho) = \bigoplus_{\bar{s}\in\mathcal{L}(\rho)} \boldsymbol{\lambda}(\bar{s})$.*

The proof is straightforward using structural induction (using Eq. 18 in the induction steps). Finally we provide the latency analysis for an FSM-SADF with a given regular expression σ. We use the same latency vector definition (Eq.20) for regular expressions where $\bar{s}$ denotes in this case, an infinite scenario sequence. We use the fact that the latency vector of an indefinite repetition of an ordinary regular expression i.e. ρ^ω directly follows from Eq.29 when k does not have an upper-bound. This enables us to provide the following computations for the latency of an FSM-SADF.

$$\boldsymbol{\lambda}(\sigma): \quad \boldsymbol{\lambda}(\rho^\omega) = (\hat{\mathbf{G}}^T(\rho))^* \boldsymbol{\lambda}(\rho) \tag{30}$$

$$\boldsymbol{\lambda}(\rho\sigma) = \boldsymbol{\lambda}(\rho) \oplus \hat{\mathbf{G}}^T(\rho)\boldsymbol{\lambda}(\sigma) \tag{31}$$

$$\boldsymbol{\lambda}(\sigma_1 + \sigma_2) = \boldsymbol{\lambda}(\sigma_1) \oplus \boldsymbol{\lambda}(\sigma_2). \tag{32}$$

Proposition 5.5. *For any expression σ, $\boldsymbol{\lambda}(\sigma)$ computed by Eqs. 30-32 satisfies $\boldsymbol{\lambda}(\sigma) = \bigoplus_{\bar{s}\in\mathcal{L}(\sigma)} \boldsymbol{\lambda}(\bar{s})$.*

The proof is straightforward using structural induction on σ. For the filter example on 9×11 frames, we have $\boldsymbol{\lambda}^T\left(((ri)^{18}((ri)^2(cm)^7(ro)^2)^9(ro)^{18})^\omega\right) = \begin{bmatrix} 53 & 51 & 2 \end{bmatrix}$, and the latency is 53 when $\boldsymbol{\gamma}_0 = \mathbf{0}$. The latency vector signifies that there are at least 53, 51 and 2 time units time differences between the output(s) that contribute to the worst-case latency (in this case the first output) and the availability time of initial tokens x, y and z respectively. This can be observed also from Fig. 3.

6 EVALUATION

We implemented the scalable throughput and latency analysis as two algorithms in the SDF3 tool [30]. We applied our analysis to dataflow models of several realistic applications listed in Table 1: H.263 decoder, MP3 decoder, down-sampler (DS), up-sampler (US), sampler (DS→US), convolution filter (CF), sub-sampled convolution filter (DS→CF→US) and multi-resolution convolution filter (MRCF). The SDF graph of the H.263 decoder is taken from Stuijk et al. [28]. This graph models the behaviour of the decoder on QCIF size images. We adapted the graph to model its behaviour for 16CIF (1408×1152) images. We applied our scalable analysis on this application by expressing its behaviour as an SADF sequence of three different modes. The MP3 decoder is modelled as an FSM-SADF [10]. To apply our analysis, we formalized its FSM by a regular expression. The convolution filter is modelled as shown in Fig. 2. The up-sampler uses a 3×3 kernel to generate four pixels out of every input pixel. The generated pixels form two pixels in one line and two pixels in the next line. To preserve the line-by-line output of data, the two bottom pixels are first stored as the kernel moves along the first line. After the first line is output, the stored pixels from the second line are output to form another complete line. The structure of operations is as follows.

(1) **frame rush-in phase**: read pixels one by one without computing any output yet for one line of the image, or W pixels;
(2) **line rush-in phase**: read one pixel without producing output;
(3) **line computation phase**: one pixel is consumed and two pixels are produced (and 2 others are stored) for a whole line minus one pixel i.e. $W - 1$ pixels. During this phase $W - 1$ pixels are consumed and $2W - 2$ pixels are produced;
(4) **line rush-out**: two pixels are produced without reading new input;
(5) **memory flush**: one line of output is flushed from memory: $2W$ pixels;
(6) repeat steps 2–5 for $H - 1$ lines;
(7) **frame rush-out phase**: produce 2 more lines of output without new input: $4W$ pixels.

Table 1. Throughput and latency analysis results for few realistic applications

Application	L	F	REM (Exact)				SOTA (Exact)		SOTA (Approx.)	
			Thr.	RT	Lat.	RT	Thr.	RT	Thr.	RT
CF (8)	37	312	0.2253	<1	49	<1	0.2253	<1	0.2025	6.00e+3
CF (32)	45	4344	0.2426	<1	145	<1	0.2426	1	0.2229	8.40e+4
CF (128)	53	66552	0.2480	<1	529	<1	0.2480	7	–	–
CF (512)	61	1.05e+6	0.2495	<1	2065	<1	0.2495	122	–	–
CF (2048)	69	1.67e+7	0.2497	<1	8209	<1	0.2497	2010	–	–
DS (512)	75	8.52e+5	9.99e−2	<1	2063	<1	9.99e−2	93	–	–
DS (1024)	81	3.40e+6	9.99e−2	<1	4111	<1	9.99e−2	360	–	–
US (128)	59	2.13e+5	0.3073	<1	271	<1	0.3073	22	–	–
US (512)	69	3.40e+6	0.3076	<1	1039	<1	0.3076	352	–	–
DS→US (2048)	98	2.66e+7	0.3076	1	18453	1	0.3076	3202	–	–
DS→CF→US (2048)	113	3.07e+7	0.2496	1	38947	1	0.2496	3695	–	–
3-level MRCF (2048)	168	7.12e+7	0.2497	1	99225	1	0.2497	8571	–	–
H.263 (16CIF)	26	3.04e+5	1.17e−8	<1	1.40e+4	1	1.17e−8	21	4.84e−10	<1
MP3 decoder	12	3169	1.71e−7	9	1.23e+7	9	1.71e−7	14	N/A	N/A
Example (Fig. 4)	13	11	0.25	1	6	<1	0.25	<1	N/A	N/A

We can use the same scenario graphs used in the model of the convolution filter (Fig. 2) with a suitable expression to express the behaviour of the up-sampler. For example the following expression models the pattern we just explained for the up-sampler.

$$\sigma_{us} = (ri)^W \left((ri)\left((cm)(ro)\right)^{W-1}(ro)^{2W+2}\right)^{H-1}(ro)^{4W}$$

For the computation phase, we used an alternation of *ro* and *cm* to express the behaviour in which one pixel is consumed and two pixels are produced. We also used *ro* for the memory flush and rush-out phases. For the down-sampler we use a 4×4 kernel to generate one pixel out of every 4 input pixels. To reduce the number of output pixels by four, the down-sampler produces an output only for every other input line and only for every other input pixel. Again if we use the same scenario graphs we used in the filter example, the pattern in which our down-sampler behaves can be described by the following expression, assuming W and H are even numbers.

$$\sigma_{ds} = (ri)^W \left((ri)^W\left((ri)(cm)\right)^{W/2}\right)^{H/2-1}(ri)^W(ro)^{W/2}$$

The sampler is an up-sampler down-sampler pipeline, where the up-sampler consumes the outputs produced by the down-sampler. In the sub-sampled convolution filter, first the image is down-sampled, then the convolution is applied to the down-sampled image and finally the filtered image is up-sampled. A block diagram of a multi-resolution filter is provided by Keinert et al. [19].

Table 1 shows the analysis results for the mentioned applications using several input image sizes. The number x following the applications denotes that the image size is $x \times x$. Parameter L is the length of the representation of the regular expression. For SDF and CSDF graphs, parameter F is the total number of actor firings within one iteration of the application, and for FSM-SADF it is the sum of all actor firings in one iteration of all scenarios. In this table the Regular Expression Method (REM) is the method provided in this paper. It uses the regular expression representation of FSM-SADF to recognize repeated sequences and transform the FSM-SADF to an FSM-SADF with a fewer number of FSM states. Then it uses the method of Geilen et al. [11] as the final throughput analysis. Moreover, it uses the same representation to compute the latency. SOTA

represents the state of the art throughput analysis methods (exact CSDF analysis [31], exact FSM-SADF analysis [11] and approximate CSDF analysis [8]). There is no existing latency analysis. The run-times (RT) reported in this table are obtained on an Ubuntu server with a 3.8Ghz processor and they are measured in milliseconds; the experiments that took more than 10 minutes were terminated, indicated with hyphens for the result and run-time. The results confirm that the REM method is scalable to applications operating on large image frames. Observe that the run-time of the state of the art methods scale linearly in F. This causes scalability problems for, for example, buffer sizing algorithms on complex applications such as multi-resolution filters. The approximate analysis often takes more than the exact analysis to terminate or it is very pessimistic. The analysis run-time for the approximate method is mainly due to transformation to a conservative SDF. According to De Groote et al., the transformation has a quadratic time complexity in the maximum number of CSDF phases, which for our examples is in the order of F [8]. We used the REM method on the MP3 decoder and the example in Fig. 4 to show the applicability of the method to FSM-SADF models. However, we do not gain much with our throughput analysis compared to the existing methods, because these models do not contain repetitive structures.

7 CONCLUSION

We provided a scalable throughput and latency analysis for multi-scale applications that are modelled by scenario-aware dataflow graphs. We showed that such models often have a cyclic behaviour with a large number of actor firings in the cycle. We overcame the scalability issue of existing exact analysis techniques by exploiting the repetitive structures within the large cycle. Our analysis scales logarithmically in the number of repetitions for such repetitive structures whereas the state of the art analysis scales at least linearly. We implemented our analysis and applied it to several realistic applications. The results show that our analysis provides accurate analysis in a shorter time compared to the existing dataflow analysis methods.

ACKNOWLEDGMENTS

This research is supported by the ARTEMIS joint undertaking through the ALMARVI project (621439).

REFERENCES

[1] H. Alizadeh Ara, A. Behrouzian, M. Geilen, et al. 2016. Analysis and Visualization of Execution Traces of Dataflow Applications. In *Proceedings of the 2nd Embedded computing and Architecture (IDEA) Workshop on Integrating Dataflow*. ESR-2017-01, 19–20.

[2] H. Alizadeh Ara, M. Geilen, T. Basten, et al. 2016. Tight temporal bounds for dataflow applications mapped onto shared resources. In *Proceedings of the 11th IEEE Symposium on Industrial Embedded Systems (SIES)*. IEEE, 1–8.

[3] François Baccelli, Guy Cohen, Geert Jan Olsder, and Jean-Pierre Quadrat. 1992. *Synchronization and linearity*. Vol. 2. Wiley New York.

[4] Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. 1999. Synthesis of Embedded Software from Synchronous Dataflow Specifications. *Journal of VLSI signal processing systems for signal, image and video technology* 21, 2 (01 Jun 1999), 151–166.

[5] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. 1996. Cyclo-static dataflow. *IEEE Trans. on signal processing* 44, 2 (February 1996), 397–408.

[6] Anne Brüggemann-Klein. 1993. Regular expressions into finite automata. *Theoretical Computer Science* 120, 2 (1993), 197 – 213.

[7] Thomas H Cormen. 2009. *Introduction to algorithms*. MIT press.

[8] R. de Groote, P. K. F. Hölzenspies, J. Kuper, and G. J. M. Smit. 2014. Single-rate Approximations of Cyclo-static Synchronous Dataflow Graphs. In *Proceedings of the 17th International Workshop on Software and Compilers for Embedded Systems (SCOPES '14)*. ACM, New York, NY, USA, 11–20.

[9] Stéphane Gaubert. 1995. Performance evaluation of (max,+) automata. *IEEE Trans. on automatic control* 40, 12 (1995), 2014–2025.

[10] M. Geilen. 2011. Synchronous Dataflow Scenarios. *ACM Trans. Embed. Comput. Syst.* 10, 2, Article 16 (Jan. 2011), 31 pages.
[11] M. Geilen, J. Falk, C. Haubelt, et al. 2014. Performance analysis of weakly-consistent scenario-aware dataflow graphs. In *Proceeding of 48th Asilomar Conference on Signals, Systems and Computers*. 393–397.
[12] M. Geilen, J. Falk, C. Haubelt, et al. 2017. Performance Analysis of Weakly-Consistent Scenario-Aware Dataflow Graphs. *Signal Processing Systems* 87, 1 (2017), 157–175.
[13] Marc Geilen and Sander Stuijk. 2010. Worst-case Performance Analysis of Synchronous Dataflow Scenarios. In *Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES/ISSS '10)*. ACM, New York, NY, USA, 125–134.
[14] A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, et al. 2006. Throughput Analysis of Synchronous Data Flow Graphs. In *Proceedings of the 6th International Conference on Application of Concurrency to System Design (ACSD'06)*. 25–36.
[15] A. H. Ghamarian, S. Stuijk, T. Basten, et al. 2007. Latency Minimization for Synchronous Data Flow Graphs. In *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*. 189–196.
[16] Michel Gondran and Michel Minoux. 1984. *Graphs and algorithms*. Wiley.
[17] C. Hagenah and A. Muscholl. 2000. Computing ϵ-free NFA from regular expressions in $\mathcal{O}(n \log^2 n)$ time. *RAIRO Inform. Théor* 34 (2000), 257–277.
[18] R. M. Karp. 1978. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics* 23, 3 (1978), 309 – 311.
[19] J. Keinert, H. Dutta, F. Hannig, et al. 2009. Model-based Synthesis and Optimization of Static Multi-rate Image Processing Algorithms. In *Proceedings of the 9th Conference on Design, Automation and Test in Europe (DATE '09)*. European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 135–140.
[20] E. A. Lee and D. G. Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (Sept 1987), 1235–1245.
[21] W. Liu, M. Yuan, X. He, et al. 2008. Efficient SAT-Based Mapping and Scheduling of Homogeneous Synchronous Dataflow Graphs for Throughput Optimization. In *Proceedings of Real-Time Systems Symposium*. 492–504.
[22] O. Moreira and H. Corporaal. 2014. *Scheduling real-time streaming applications onto an embedded multiprocessor*. Springer, 67–75.
[23] A. Nelson, K. Goossens, and B. Akesson. 2015. Dataflow formalisation of real-time streaming applications on a Composable and Predictable Multi-Processor SOC. *Systems Architecture* 61, 9 (2015), 435 – 448.
[24] P. Poplavko, T. Basten, M. Bekooij, et al. 2003. Task-level Timing Models for Guaranteed Performance in Multiprocessor Networks-on-chip. In *Proceedings of 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '03)*. ACM, New York, NY, USA, 63–72.
[25] S. Ritz, M. Pankert, V. Zivojinovic, and H. Meyr. 1993. Optimum vectorization of scalable synchronous dataflow graphs. In *Proceedings of 1993 International Conference on Application-Specific Array Processors*. 285–296.
[26] F. Siyoum, M. Geilen, and H. Corporaal. 2014. Symbolic analysis of dataflow applications mapped onto shared heterogeneous resources. In *Proceedings of the 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6.
[27] S. Sriram and S. S. Bhattacharyya. 2009. *Embedded multiprocessors: Scheduling and synchronization*. CRC press.
[28] S. Stuijk. 2007. *Predictable mapping of streaming applications on multiprocessors*. Ph.D. Dissertation. Eindhoven, NL.
[29] S. Stuijk, M. Geilen, and T. Basten. 2006. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *Proceedings of the 43rd ACM/IEEE Design Automation Conference*. 899–904.
[30] S. Stuijk, M. Geilen, and T. Basten. 2006. SDF3: SDF for free. In *Proceedings of the 6th International Conference on Application of Concurrency to System Design*. IEEE, 276–278.
[31] S. Stuijk, M. Geilen, and T. Basten. 2008. Throughput-Buffering Trade-Off Exploration for Cyclo-Static and Synchronous Dataflow Graphs. *IEEE Trans. Comput.* 57, 10 (Oct 2008), 1331–1345.
[32] S. Stuijk, M. Geilen, B. Theelen, and T. Basten. 2011. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*. 404–411.
[33] B. D. Theelen, M. C. W. Geilen, T. Basten, et al. 2006. A Scenario-aware Data Flow Model for Combined Long-run Average and Worst-case Performance Analysis. In *Proceedings of the 4th ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE '06)*. IEEE Computer Society, Washington, DC, USA, 185–194.