Throughput-Buffering Trade-Off Analysis for Scenario-Aware Dataflow Models

Hadi Alizadeh Ara
Eindhoven University of Technology
Eindhoven, The Netherlands
S.H.Seyyed.Alizadeh@tue.nl

Marc Geilen
Eindhoven University of Technology
Eindhoven, The Netherlands
M.C.W.Geilen@tue.nl

Amir Behrouzian
Eindhoven University of Technology
Eindhoven, The Netherlands
A.R.Baghban.Behrouzian@tue.nl

Twan Basten
Eindhoven University of Technology
& ESI, TNO
Eindhoven, The Netherlands
A.A.Basten@tue.nl

ABSTRACT

In multi-media applications, buffers represent storage spaces that are used to store the data communicated between different tasks in the application, and throughput refers to the rate at which output data is produced by the application. The capacities of the buffers influence the throughput, by altering the waiting times for tasks that need to read or write data from or to the buffers. The buffers are realized using memory. To minimize the memory usage, we look for algorithms to compute the minimal capacity requirements for buffers to execute an application under a given throughput constraint. Synchronous dataflow (SDF) is a common formalism used to model applications in such algorithms. SDF however, is not suitable to describe today's dynamic applications, as it cannot express task variations. Finite-State-Machine Scenario-Aware Dataflow (FSM-SADF) is an extension of SDF that allows for not only task variations, but also structural variations, making it suitable for a wide range of dynamic applications. This paper provides the first throughput-buffering trade-off analysis for FSM-SADF models. The analysis provides the Pareto space of throughput and storage space trade-offs. The trade-off analysis is done by a guided Design Space Exploration (DSE) that cuts-off the exploration on non-critical buffers. The core of such a DSE is an FSM-SADF throughput analysis that, given the capacity of every buffer, obtains the throughput, as well as the critical buffers. We demonstrate the feasibility of our analysis with a number of examples.

ACM Reference Format:
Hadi Alizadeh Ara, Marc Geilen, Amir Behrouzian, and Twan Basten. 2018. Throughput-Buffering Trade-Off Analysis for Scenario-Aware Dataflow Models. In *26th International Conference on Real-Time Networks and Systems (RTNS '18), October 10–12, 2018, Poitou, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3273905.3273921

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
RTNS '18, October 10–12, 2018, Poitou, France
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6463-8/18/10…$15.00
https://doi.org/10.1145/3273905.3273921

1 INTRODUCTION

Applications used for multi-media streaming and wireless communications in embedded devices need real-time computing. In particular, these applications require that the computations involved in them are carried out at a certain rate, referred to as throughput. The throughput in a video application running on a smart phone for instance, determines the output frame rate, which is a measure of performance or quality in such applications. Nowadays, with the increasing complexity of these applications and the need for higher quality embedded devices, the amount of computations required for these applications is more than the processing capabilities of a single processor to keep up with the required throughput. Therefore, multiprocessor systems are used to share the computational load. The inherent task parallelism in concurrent applications is exploited to deliver the required performance [9].

H.263 is a video compression standard used in multi-media and video-conferencing. An H.263 decoder is an application that decodes the compressed frames and transforms them into a format recognizable by embedded devices. The frames require the following tasks to be decoded [15]: Variable Length Decoding (VLD), Inverse Quantization (IQ), Inverse Discrete Cosine Transformation (IDCT), Motion Compensation (MC) and Reconstruction (RC)[1]. The VLD kernel executes on every frame and decomposes it into a number of small blocks of data items called macro blocks. The macro blocks are stored in a buffer dedicated to store the outputs of VLD. The IQ and IDCT kernels read the macro blocks from the buffer one by one, decode them and store them in another buffer. When all macro blocks are decoded, the MC and RC execute to reconstruct the decoded frame from the decoded macro blocks. Usually the decomposition, the macro block computations and the reconstruction tasks are each mapped to a separate processor to exploit the concurrency. To obtain the maximum throughput, the tasks are scheduled to start as soon as possible, i.e., as soon as they have enough input data and enough space to produce their outputs.

The capacities of the buffers influence the throughput of the application by altering the task schedules. In the H.263 decoder, the buffer that stores the outputs of the VLD must have enough capacity to store the macro blocks decomposed from one frame, otherwise the application deadlocks because VLD will never start. The number

[1] A list of abbreviations can be found in Appendix A.

of macro blocks produced by an execution of VLD depends on the frame size. If the buffer can store exactly this number of macro blocks, the VLD and IQ always execute sequentially even though they are mapped to different processors. This is because VLD will not start decomposing the next frame until all macro blocks in the buffer are decoded and the buffer is emptied. By increasing the capacity of the buffer by one macro block, the VLD on the next frame can run in parallel with one (the last) IQ execution in the current frame. By further increasing the capacity, more IQ executions will be executed in parallel with VLD, as VLD takes 50 times as much time to complete as a single execution of IQ. Increasing the capacity macro block by macro block, will increase the throughput, assuming that this buffer is the limiting factor for the throughput (i.e., when other tasks have shorter execution times compared to VLD and other buffers are large enough). If we keep increasing the capacity, at some point the VLD on the next frame will be fully parallelized with IQ executions. From this moment, increasing the capacity of this buffer will not affect the throughput as the application has reached its maximum throughput.

The capacities of the buffers should be be optimized for the required throughput, since the buffers are implemented using costly and power consuming memory blocks. To formalize the buffer sizing problem, applications are mathematically described by a Model of Computation (MoC). Synchronous Dataflow (SDF) [11] is a MoC that is used in a number of buffer sizing techniques. In SDF, it is possible to model buffers with a limited capacity. This is meaningful when modelling and analyzing implementations of the signal processing applications. The buffers in applications such as H.263 decoder, sample rate converter and modem are sized using SDF models [15][1]. SDF abstracts the tasks by actors. Actor firings mimic task executions; they consume input data, take a constant time and then produce output data. Actors start firing as soon as there is enough input data and buffer space to execute the tasks they represent. In dataflow, buffers are modelled by channels located between actors. Actor firings produce tokens on these channels. The tokens represent data items that take up a certain amount of space in memory. This way, the dataflow semantics describes the execution behaviour of the applications. SDF models can be efficiently analysed for their throughput and therefore, they are employed in iterative design procedures to find optimal buffer sizes [8][7][15].

Modern applications are dynamic. Dynamic applications include tasks with behaviour that changes at run-time. SDF actors are unable to represent task variations and consequently, they are not suitable for modelling dynamic applications. For instance, a task that alternatively switches between complete and partial executions can not be represented by an SDF actor. Even though a conservative model can be obtained by considering the complete execution at all times, the analysis results will be pessimistic and may result in memory over-allocation [15]. Cyclo-Static Dataflow (CSDF) [2] is a generalization of SDF. CSDF actors can represent task variation with periodic pattern that is known a priori, such as the variable task mentioned above. However, many modern applications have non-deterministic task variation patterns and/or their variation is at the level of the graph structure rather than being at the task level [6].

As an example, let us look at an MPEG-4 Simple Profile decoder [6]. Similar to the H.263 decoder, the MPEG-4 decoder has a VLD that decompresses the encoded input frames into a stream of macro blocks. In this decoder however, the frames are classified in I-frames and P-frames. Decoding of I and P-frames is partly different as they are encoded differently. I-frames are encoded independently from previous frames, but P-frames need a reference to a previous frame as they exploit temporal correlation in the form of motion vectors between subsequent frames. Therefore, the decomposed macro blocks from a P-frame are decoded and then made into a decoded frame using both MC (motion compensation) and RC (reconstruct). I-frame macro blocks lacking motion vectors are reconstructed using RC only. MPEG-4 is a dynamic application with behaviour that changes based on the input frame type and the number of motion vectors present, which is not deterministic. Moreover, the structure of the application changes when decoding I and P-frames. For example, MC is not needed for I-frames while it is a part of the decoding process for P-frames.

Finite-State-Machine Scenario-Aware Dataflow (FSM-SADF) [6] is an extension of SDF and CSDF that can handle enough dynamism to model applications such as the MPEG-4 decoder, while still being amenable to analysis. In this model, scenarios represent modes of operation. For instance, MPEG-4 has naturally two modes of operation, namely I-frame and P-frame modes. In the P-frame mode, moreover, actor reading and writing rates and their execution times change between executions due to different numbers of motion vectors present in the data. In FSM-SADF, modes are described by (parameterized) SDF models. FSM-SADF provides additional semantics that captures parametric rates and execution times and switching between scenarios. A Finite State Machine (FSM) is used to specify switching patterns, which can be deterministic or non-deterministic.

Finding the minimal buffer sizes for an SDF model to satisfy a minimum throughput constraint is known to be a complex problem (NP-complete [12]). Nevertheless, Stuijk et al. [15] provide a throughput-buffering trade-off analysis that terminates in a reasonable time for SDF and CSDF models of real-life applications. The algorithm uses a Design Space Exploration (DSE) scheme that prunes the design space (without losing any optimal points) during the exploration. The approach uses a throughput analysis algorithm to compute the throughput of the model, and at the same time find the throughput limiting buffers. The exploration continues only on these buffers. This makes the exploration space smaller and smaller as the exploration continues.

The trade-off analysis of Stuijk et al. [15] is not directly applicable to FSM-SADF models. Even though every scenario can be separately sized for its buffers using the SDF techniques, the results will be useful only in a corner case where the scenarios do not share any actors or buffers. For instance, if the I and P-frame decoding tasks in the MPEG-4 decoder are implemented as two separate applications. This is often not the case, because it increases the resource usage and reduces the maintainability of the application. Dynamic reading and writing patterns on buffers lead to different requirements compared to the strictly regular patterns occurring in SDF

In this paper, we provide the first throughput-buffering trade-off analysis for applications modelled as FSM-SADF. As an enabling contribution, we propose a novel throughput computation method for FSM-SADF models that obtains the throughput and at the same time determines the buffers that limit the throughput of the model.

This is an important contribution because it enables the application of smart exploration schemes in performing efficient buffer sizing for FSM-SADF models. The second contribution realizes the main goal of this paper by integrating our throughput analysis in a DSE scheme similar to the one provided by Stuijk et al. [15] to find all optimal throughput-buffering trade-offs. The DSE is tuned to work with FSM-SADF models. We prove that the proposed DSE finds all optimal points in the space of the throughput-buffer size trade-offs. We show the applicability of our approach using several realistic applications modelled as FSM-SADF.

2 RELATED WORK

The problem of finding the minimum buffer requirements to execute an SDF with the maximum, minimum or a given throughput has been addressed in a number of works [8][7][15][18]. Hwang et al. [8] and Govindarajan et al. [7] formalize the maximum throughput version of the question as a linear programming problem. The formalization provided by Hwang et al. [8] consider time-constrained and resource-constrained schedules and it is applicable to acyclic SDF only. The authors of [7] provide a heuristic approach to solve the problem for SDF, as the problem is NP-complete [12]. The approach taken by Stuijk et al. [15] is based on a DSE. Their approach finds all trade-offs (Pareto points) in the space of throughput-buffer size by using a guided exploration rather than an exhaustive exploration. The exploration is narrowed down to the buffers that create the so-called storage dependencies while the SDF is executing with a certain throughput.

Buffer minimization techniques for CSDF graphs are presented in several works [17][4][3][15]. Denolf et al. [4] present an approach to find the minimum buffers required to execute a CSDF graph with a static schedule. However, there is no guarantee that another schedule that realizes the same throughput with smaller buffer requirements does not exist. A heuristic algorithm given by Wiggers et al. [17] minimize buffer usage under a given throughput. Bodin et al. [3] provide a linear programming formalization of the minimum buffer requirements under a throughput constraint. The authors give an algorithm to solve the problem approximately.

Our work extends the exact throughput-buffering trade-off analysis for SDF and CSDF provided by Stuijk et al. [15] to FSM-SADF graphs. We use a guided exploration of the design space by identifying critical buffers, similar to the so-called storage dependencies identified from the state-space of the SDF/CSDF execution [15].

3 PRELIMINARIES

3.1 SDF

SDF describes the application by a directed graph. The left side of Figure 1 shows an example Synchronous Dataflow Graph (SDFG). In this graph, nodes depict actors. Actors represent individual tasks and actor firings correspond to task executions. The letters inside the nodes are used to refer to actors and the numbers next to the letters show the execution times of the actors. Directed edges in the graph represent channels and they are referenced by the annotations below them. Channels enforce dependencies between actor firings caused by, for instance, data dependencies or control decisions. Channels may initially contain tokens, referred to as initial tokens. In Figure 1, initial tokens are shown by black spots on top of the channels and they are referenced by annotations next to them. When an actor starts firing, it consumes tokens from its input channels, and when its execution time expires, it produces tokens on its output channels. All firings of an actor consume and produce constant numbers of tokens per channel, called the consumption and production rates. The rates are given as edge annotations in the graph (rates of 1 are not shown to avoid cluttering). For instance, every firing of actor P produces two tokens on channel a.

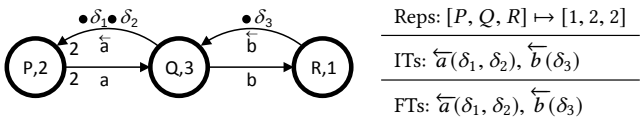


Figure 1: An example SDF scenario

The channels that model data dependencies are called buffer channels. In Figure 1, channels a and b are buffer channels. SDF channels do not have control over the number of tokens that accumulates on them. Therefore, to model a limited capacity buffer with back-pressure, it is common to put a channel in the opposite direction between the two actors with a number of tokens on it [15]. We call these channels capacity control channels or capacity channels for short (channels $\overleftarrow{a}$ and $\overleftarrow{b}$ in the figure). The capacity of the buffer channel is the sum of the tokens on the buffer channel and the tokens on the capacity channel. In Figure 1, there is a buffer with 2 tokens capacity between P and Q and a buffer with 1 token capacity between Q and R.

An SDFG is defined by a tuple (A, C). The set A contains a finite number of actors. An SDF graph defines a worst-case execution time $e(a) \in \mathbb{R}^+$ for every actor $a \in A$. The set $C \subseteq A \times A$ is the set of channels. Let $c \in C$ be a channel from an actor $P \in A$ to an actor $Q \in A$ (P and Q can be the same actor). Channel c is an output channel of P and an input channel of Q. Each of actors P and Q respectively associate a non-negative number to channel c, namely the output rate $Out_P(c)$ and input rate $In_Q(c)$. Channel c contains a number of initial tokens $It(c)$. We consider a set $B \subseteq C$ of buffer channels with limited capacities for an SDFG. Let $b \in B$ be a buffer channel from an actor P to an actor Q ($P \neq Q$). The SDFG includes a capacity channel $\overleftarrow{b}$ from Q to P, such that $Out_Q(\overleftarrow{b}) = In_Q(b)$ and $In_P(\overleftarrow{b}) = Out_P(b)$.

3.2 SDF Scenario

SDF scenarios describe deterministic SDF behaviours. An SDF scenario specifies a finite non-empty set of actor firings in an SDFG, which is often represented by a vector that shows the exact number of times (repetitions) that each actor in the graph fires. Figure 1 shows an example SDF scenario. The table on the right side of this figure shows the repetition vector for the scenario. In the example scenario, the actor repetitions correspond to one iteration of the example SDFG. An iteration of an SDFG is the smallest, non-empty set of actor firings that does not change the number of tokens on the channels. For the example SDFG, one iteration contains one firing of actor P and two firings of actors Q and R each. In general, a scenario may correspond to a partial iteration. For instance, it is possible to define a scenario that includes only one execution

of P, i.e. $[P, Q, R] \mapsto [1, 0, 0]$. Such scenarios are introduced by Geilen et al. [5]. Our work considers the general case.

Executing a scenario is to execute all actor firings defined by the scenario. A scenario execution may deadlock due to insufficient buffer capacities. For instance, if buffer a had a capacity of 1 token (assume only one token on $\overleftarrow{a}$), the example scenario deadlocks, because firings of P need to consume 2 tokens from channel $\overleftarrow{a}$ (corresponding to claiming two output spaces in buffer channel a), so P can not fire and no actor will be able to fire anymore.

A deadlock-free scenario execution corresponds to a certain amount of real-world progress. The progress can be associated with different meanings depending on the application. In the MPEG-4 decoder model given by Geilen et al. [6], the unit of progress is defined as the decoding of one frame. In their model, both I and P-frame scenarios have a progress of 1, as the actor firings included in those scenarios correspond to the task executions required for decoding one frame. If the designer is interested in the number of firings of a specific actor, the repetition of that actor can serve as the progress of the scenario. For simplicity and without loss of generality we assume that all scenarios have a progress of 1.

A deadlock-free scenario execution leaves a number of tokens on the channels, called final tokens. Initial and final tokens (ITs and FTs) in a scenario are labelled. In Figure 1, for instance, δ_3 labels the initial token on channel $\overleftarrow{b}$ and the final token produced on the same channel after the scenario is executed. ITs an IFs in this figure show the initial and final token labels for all channels respectively. ITs and IFs are a means to convey (timing) information from an executed scenario to the scenario that will be executed next. The final tokens of the executed scenario take the place of the initial tokens with the same labels in the next scenario. If the ITs and IFs in a scenario transition do not match then the model is incorrect. In our FSM-SADF examples, we allow transitions from a scenario to itself. This means that ITs and IFs should match within the scenario, i.e. they should be the same. If we do not allow self-transitions, then ITs and IFs can be different.

3.3 (max,+) Representation of an SDF Scenario

Scenario executions can be described in (max, +) algebra. We first introduce the notations for this algebra. (max, +) algebra defines two binary operators, maximum ($u \oplus v = \max\{u, v\}$) and addition ($u \otimes v = u + v$), where $u, v \in \mathbb{R}_{-\infty} = \mathbb{R} \cup \{-\infty\}$. Let vectors $\boldsymbol{v}$ and $\boldsymbol{u}$ belong to the n dimensional vector space denoted by $\mathbb{R}^n_{-\infty}$. The inner product of $\boldsymbol{u}$ and $\boldsymbol{v}$ is defined as $\boldsymbol{u}^T\boldsymbol{v} = (u_1 \otimes v_1) \oplus (u_2 \otimes v_2) \oplus \cdots \oplus (u_n \otimes v_n)$. Operator $\oplus$ on vectors with the same size is an element-wise operator. Matrix multiplications and additions are defined using the inner product and addition of vectors defined as above. For the sake of readability we do not write $\otimes$ with matrix/vector multiplications. The norm of a vector $\|\boldsymbol{v}\|$ is equal to the maximum entry in the vector. The element on the i^{th} row of the j^{th} column of a matrix $\mathbf{G}$ is denoted by $[\mathbf{G}]_{i,j}$.

The timing relations between the time-stamps of the tokens consumed and produced by an actor firing in an SDF scenario are naturally captured by maximization and addition operations. For instance, consider the firing of actor P in the example scenario. Let t_{δ_1} and t_{δ_2} denote the time-stamps (availability times) of initial tokens δ_1 and δ_2 respectively and, t denote the time-stamp (production time) of tokens produced by the firing of P. The firing of P starts at $\max\{t_{\delta_1}, t_{\delta_2}\}$ and, completes two time units after the start. Therefore, equation $t = \max\{t_{\delta_1}, t_{\delta_2}\} + 2$ describes the relation between the time-stamps of the produced and consumed tokens in this firing. This relation corresponds to a linear equation in (max, +) algebra. For instance, the equation above can be written as $t = 2 \otimes t_{\delta_1} \oplus 2 \otimes t_{\delta_2}$ which is a linear equation in (max, +). Consequently, deadlock-free executions of an SDF scenario can be described using a set of linear equations that relate the time-stamps of the final tokens to the time-stamps of the initial tokens. These equations can be compactly represented using a matrix vector-multiplication as follows.

$$\mathbf{G} = \begin{array}{c} \begin{matrix} \delta_1 & \delta_2 & \delta_3 \end{matrix} \\ \begin{bmatrix} 5 & 5 & 3 \\ 9 & 9 & 7 \\ 10 & 10 & 8 \end{bmatrix} \end{array} \begin{matrix} \delta_1 \\ \delta_2 \\ \delta_3 \end{matrix}$$

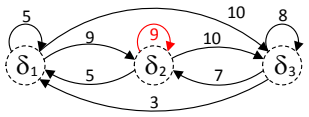


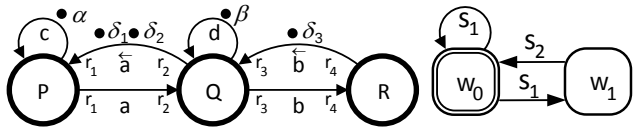
Figure 2: The matrix of the example scenario and the MPA of the example FSM-SADF

$$\boldsymbol{\gamma}' = \mathbf{G}(s)\boldsymbol{\gamma} \tag{1}$$

In Eq. 1, $\boldsymbol{\gamma}$ and $\boldsymbol{\gamma}'$ are vectors that contain the time-stamps of the initial and final tokens in a scenario s respectively (also called state vectors) and $\mathbf{G}(s)$ is the (max,+) matrix of scenario s. The matrix $\mathbf{G}$ in Figure 2 represents the example scenario. The timing relations of tokens labelled δ_1, δ_2 and δ_3 in Figure 1 are captured in the first, second and third column/row of this matrix, respectively. For example, the first row of the matrix indicates that there are at least 5, 5 and 3 time units delay between the respective time-stamps of initial tokens δ_1, δ_2 and δ_3 and the time-stamp of final token δ_1.

A symbolic actor firing [6] is represented by a vector dot-product. Again consider the firing of actor P. The time-stamps of the tokens produced by this firing can be described as $t = \boldsymbol{g}^T\boldsymbol{t} = [\ 2 \quad 2 \quad -\infty\ ][\ t_{\delta_1} \quad t_{\delta_2} \quad t_{\delta_3}\ ]^T$. Vector $\boldsymbol{t}$ contains the time-stamps of the initial tokens. Vector $\boldsymbol{g}$ contains the time added to the time-stamp of every produced token by the firing of actor P. The $-\infty$ entry captures the fact that firing P does not have an effect on the time of token δ_3.

A (max, +) matrix of a scenario can be computed by symbolically simulating all actor firings in the scenario [6]. The initial tokens δ_1, δ_2 and δ_3 are respectively assigned with $\boldsymbol{g}_{\delta_1} = [\ 0 \quad -\infty \quad -\infty\ ]$, $\boldsymbol{g}_{\delta_2} = [\ -\infty \quad 0 \quad -\infty\ ]$ and $\boldsymbol{g}_{\delta_3} = [\ -\infty \quad -\infty \quad 0\ ]$. Firing actor P requires initial tokens δ_1 and δ_2 and, it produces two tokens on channel a with time-stamp vector $\max\{\boldsymbol{g}_{\delta_1}, \boldsymbol{g}_{\delta_2}\} + 2 = \max\{[\ 0 \quad -\infty \quad -\infty\ ], [\ -\infty \quad 0 \quad -\infty\ ]\} + 2 = [\ 2 \quad 2 \quad -\infty\ ]$. The first firing of actor Q consumes a token from channel a and initial token δ_3 and produces the final token δ_1 and a token on channel b with time-stamp vector $\max\{[\ 2 \quad 2 \quad -\infty\ ], [\ -\infty \quad -\infty \quad 0\ ]\} + 3 = [\ 5 \quad 5 \quad 3\ ]$. By simulating the rest of the actor firings we obtain the time-stamps vectors of all final tokens. By vertically augmenting the time-stamp vectors of the final tokens as $[\boldsymbol{g}_{\delta_1}; \boldsymbol{g}_{\delta_2}; \boldsymbol{g}_{\delta_3}]$, we obtain the matrix shown in Figure 2.



(a) Scenario graphs **(b) FSM**

Scs	s_1	s_2
ETs	$e(P) = 2, e(Q) = 3, e(R) = 1$	$e(P) = 1, e(Q) = 5, e(R) = 2$
Rates	$r_1 = 3, r_2 = 4, r_3 = 2, r_4 = 6$	$r_1 = 1, r_2 = 2, r_3 = 4, r_4 = 6$
Progs	1	1
Reps	$[P, Q, R] \mapsto [4, 3, 1]$	$[P, Q, R] \mapsto [6, 3, 2]$
ITs	$\overleftarrow{a}(\delta_1, \delta_2)$, $\overleftarrow{b}(\delta_3)$, $c(\alpha)$, $d(\beta)$	$\overleftarrow{a}(\delta_1, \delta_2)$, $\overleftarrow{b}(\delta_3)$, $c(\alpha)$, $d(\beta)$
FTs	$\overleftarrow{a}(\delta_1, \delta_2)$, $\overleftarrow{b}(\delta_3)$, $c(\alpha)$, $d(\beta)$	$\overleftarrow{a}(\delta_1, \delta_2)$, $\overleftarrow{b}(\delta_3)$, $c(\alpha)$, $d(\beta)$

(c) Scenario specifications

Figure 3: An FSM-SADF with two scenarios

3.4 FSM-SADF

FSM-SADF is a variant of dataflow models that expresses dynamic SDF behaviours by switching between several SDF scenarios. Figure 3 shows an FSM-SADF with two scenarios, namely s_1 and s_2. Figure 3a shows the scenario graphs of these scenarios and Table 3c shows their specifications. The structure of the scenario graphs are the same, but the actor execution times and some rates are different for every graph. Without loss of generality we assume that a buffer with a certain capacity in one scenario graph is also a buffer with the same capacity in all other scenario graphs. In the scenario graphs of Figure 3, channels a and b are buffer channels with capacities 2 and 1, respectively. To ensure that a buffer capacity that is occupied by a scenario cannot be used in an other scenario at the same time, the tokens that model the capacity of the buffers are assigned with the same labels in all scenarios. Observe that δ_1, δ_2 and δ_3 label the same tokens in both scenario graphs.

In FSM-SADF, a particular behaviour is achieved by executing a sequence of scenarios. When executing a sequence of scenarios, the execution dependencies among scenarios are established through the initial and final tokens. When the FSM-SADF in Figure 3 executes the scenario sequence $s_1 s_2$, it executes scenario s_2 after s_1, the time-stamp vector of the final tokens in scenario s_1 becomes the time-stamp vector of the initial tokens with the same labels in scenario s_2. FSM-SADF uses an FSM to specify all possible behaviours described as scenario sequences. Figure 3b shows the FSM of the FSM-SADF in Figure 3. An FSM-SADF is said to be consistent if for any scenario sequence $\bar{s} = s_1 s_2 \cdots$ recognized by FSM, the number of final tokens and their labels in s_i match the number of initial tokens and their labels in s_{i+1}.

An FSM-SADF is defined by a tuple $(S, g, r, i, f, p, \mathcal{A})$ [6]. The set S contains a finite number of scenarios. For every scenario $s \in S$, $g(s)$ maps s to an SDFG, $r(s)$ defines a vector of actor repetitions for s, $i(s)$ is the number of initial tokens, $f(s)$ is the number of final tokens and $p(s)$ maps s to a non-negative real number that corresponds to the amount of progress in executing s. The FSM $\mathcal{A}$ is a tuple (W, w_0, λ) with a set W of states, an initial state w_0 and a labelled transition relation $\lambda \subseteq W \times S \times W$. To ensure a consistent FSM-SADF, for any state $w \in W$, any incoming edge labelled with scenario s_i and outgoing edge labelled with scenario s_{i+1}, $f(s_i) = i(s_{i+1})$, i.e., the number of final and initial tokens of subsequent scenarios must match. This number of tokens for a state w is denoted by $n(w)$.

We define a set U of *common buffers* for an FSM-SADF. Every common buffer $u \in U$ corresponds to a buffer channel in the scenario graph of every scenario $s \in S$. For an FSM-SADF, we define functions $b_s(u)$ to map every common buffer $u \in U$ to the corresponding buffer channel in $g(s)$ of every scenario $s \in S$. Similarly, we define functions $c_s(u)$ that map every buffer $u \in U$ to the capacity channel of its corresponding buffer in $g(s)$. For the FSM-SADF in Figure 3, we define $U = \{u_a, u_b\}$. Common buffers u_a and u_b are respectively mapped to buffer channels a and b in both scenarios.

3.5 Modelling Buffer Requirements

To be able to execute an FSM-SADF, we need to allocate storage space and distribute it over its buffers. To formalize the distribution of the storage space over the buffers in an FSM-SADF, we define storage distributions. Similar to the definition provided by Stuijk et al. [15], a storage distribution of an FSM-SADF on a set U of common buffers is a mapping $d : U \rightarrow \mathbb{N} \cup \{0\}$ that associates, with every buffer $u \in U$ the capacity of the buffer. The allocated storage space, i.e. the size of the distribution, is defined as $|d| = \sum_{u \in U} d(u)$. For simplicity and without loss of generality, in this definition we assume all tokens represent the same amount of storage space. This can be easily generalized by considering different weights for tokens in different buffers. We write $d_1 \preceq d_2$ if and only if for every $u \in U$, $d_1(u) \leq d_2(u)$. In the reminder, an FSM-SADF F that incorporates a storage distribution d is denoted by F_d. In F_d, for every $u \in U$, there exists $d(u) - It(b_s(u))$ tokens on channel $c_s(u)$ in every scenario graph $g(s)$. The FSM-SADF shown in Figure 3 incorporates a distribution d that allocates two tokens to u_a and one token to u_b, i.e., $d(u_a) = 2$ and $d(u_b) = 1$. We denote this distribution by $\langle u_a, u_b \rangle \mapsto \langle 2, 1 \rangle$. The size of this distribution is 3 tokens.

3.6 FSM-SADF Throughput Analysis

Let $F_d(S, g, r, i, f, p, \mathcal{A})$ be an FSM-SADF that incorporates a storage distribution d. If d is such that at least one of the scenarios deadlocks, then throughput is defined zero. Considering deadlock-free scenarios, the throughput of a scenario sequence is defined as the average amount of progress made per time unit during the execution of that sequence. The throughput of every $\bar{s} = s_1 s_2 \cdots$ recognized by $\mathcal{A}$ is defined as follows [6].

$$Thr(\bar{s}) = \lim_{i \to \infty} \sup \frac{\Sigma_{n=1}^{i} p(s_n)}{\|\boldsymbol{\gamma}_i\|}. \tag{2}$$

In Eq. 2, $\boldsymbol{\gamma}_i$ denotes the time-stamp vectors of the initial tokens in scenario s_i. The throughput of an FSM-SADF is the worst case throughput obtained among the set of all scenario sequences in the language of $\mathcal{A}$ i.e. $L(\mathcal{A})$, as follows.

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

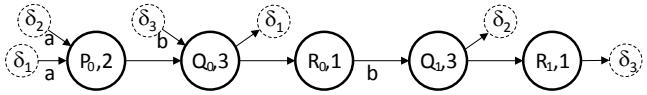


Figure 4: The dependency graph of the example scenario

A throughput computation method based on this definition is provided by Geilen et al. [6]. This method first computes the (max, +) matrix $\mathbf{G}(s)$ of every scenarios $s \in S$. Then these matrices are used to generate a (max, +) Automaton (MPA) defined in the form of a graph $M(R, E)$ with nodes R and edges E, as follows.

$$R = \{(w, i) \mid w \in W, 1 \leq i \leq n(w)\}$$

$$E = \Big\{((w_1, i), [\mathbf{G}(s)]_{i,j}, p(s), (w_2, j)) \mid (w_1, s, w_2) \in \lambda, 1 \leq i \leq n(w_1), 1 \leq j \leq n(w_2), [\mathbf{G}(s)]_{i,j} \neq -\infty\Big\}$$

We explain this by an example. To have a simple FSM-SADF example to work with, we consider an FSM-SADF that repeatedly executes the scenario in Figure 1. We refer to the example as the example FSM-SADF in the reminder of the paper. Figure 2 shows the MPA of the example FSM-SADF. This graph contains 3 nodes, one node for every initial token. Every edge in this MPA corresponds to an element in the matrix of the example scenario. For instance, the edge from δ_3 to δ_2 corresponds to the element in the third column, second row of the matrix and it has a delay of 7 time units. Every edge has a progress of 1 unit (not shown in the figure to avoid cluttering). The throughput is determined using a Maximum Cycle Ratio (MCR) analysis on the MPA [6][5]. The ratio of a cycle equals the sum of the weights on its edges, divided by the number of edges in the cycle. The weight of an edge equals the incurred delay divided by its progress. The MCR of the MPA of the example FSM-SADF is 9/1. Therefore the throughput is 1/9 unit of progress per time unit. Critical cycles of an MPA are the cycles with the maximum cycle ratio. The MPA in Figure 2 has one critical cycle, which is coloured in red.

4 THE MAIN IDEA

We use a DSE to find all throughput-buffering trade-offs for applications modelled as FSM-SADF. The DSE method requires that for a given storage distribution, the buffers that limit the throughput, are identified. The throughput analysis of Geilen et al. [6] does not give an indication of the buffer channels with storage dependencies, as (max, +) representations leave out the notion of actors and channels. Therefore, it cannot be directly used in the DSE. Observe that there is no obvious way to identify the throughput limiting buffers from the critical cycle of the MPA. In Figure 2, the critical cycle goes through δ_2, i.e., δ_2 is the only critical token. This gives a false impression that a is the only limiting buffer (because δ_2 is on a) and therefore, increasing the capacity of buffer b should not lead to a higher throughput. However, our experiments show that in fact, both a and b are limiting buffers and increasing the capacity of b by one token leads to a higher throughput compared to when the capacity of a is increased by two tokens.

In this paper, we arm the traditional (max,+) matrices with information on the buffers. The MPA built from the new matrices carries

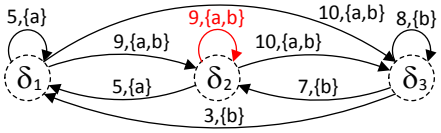


Figure 5: The EMPA of the example scenario

the buffer information on every state transition. This enables us to translate the critical cycles in the MPA back to a set of buffer channels we refer to as *critical buffer channels*. This is done by extracting additional information from the simulation when transforming an SDF scenario to its matrix representation. We explain this by defining a dependency graph of a scenario. The dependency graph shows the immediate dependencies between actor firings defined by the scenario. The dependency graph of the example scenario is shown in Figure 4. In this graph large nodes represent actor firings. A directed edge from firing A_i to firing B_j indicates that the j^{th} firing of actor B can immediately start when the i^{th} firing of actor A is completed. If an edge represents a firing dependency that is caused by a capacity channel, the corresponding buffer is mentioned under the edge. The production and consumption of the initial tokens by the firings are shown by smaller nodes. Observe that the critical path (from δ_2 to δ_2) goes through buffers a and b, which means that they are both critical.

By backtracking the (longest) path from the production of every token to the consumption of every token, we can obtain matrix $\mathbf{G}$, as well as the set of buffers that affect every element of this matrix. The results is the Extended Matrix Representation (EMR) of the SDF scenario where every element of the matrix is a pair that contains the time-stamp component and the buffer set component. Such a matrix for the example scenario is as follows.

$$\mathbf{G}^e = \begin{bmatrix} (5, \{a\}) & (5, \{a\}) & (3, \{b\}) \\ (9, \{a, b\}) & 9, \{a, b\} & (7, \{b\}) \\ (10, \{a, b\}) & (10, \{a, b\}) & (8, \{b\}) \end{bmatrix}$$

Observe that, for instance, buffer b affects every path backtracked from δ_2 and δ_3 in Figure 4. Therefore, the buffer set components of the elements in the second and third row of the matrix $\mathbf{G}^e$ include b. The MPA can be built from the EMR in a similar way as it is built from the (max, +) representation, except that now the MPA edge labels contain time-stamp components and buffer set components. We refer to such an MPA as the Extended (max, +) Automaton (EMPA). The EMPA of the example FSM-SADF is shown in Figure 5. Since a and b are on the critical cycle, they are identified as critical.

In Section 5, we extend the symbolic simulation method provided by Geilen et al. [6] to directly generate the EMR of scenarios (without the need to first create the dependency graph). Section 6 defines the critical buffers in FSM-SADF graphs. In Section 7, we provide our throughput-buffering trade-off analysis for FSM-SADF.

5 THE EXTENDED MATRIX REPRESENTATION

This section introduces the extended matrix representation of an SDF scenario. Then it provides an algorithm to compute it for a given deadlock-free SDF scenario. Consider a scenario with a set

B of buffer channels in its scenario graph. The elements of the EMR are pairs $(t, \tilde{B})$. Component $t \in \mathbb{R}^{+}_{-\infty}$ indicates the relation between the time-stamps of the initial and final tokens (as in the traditional matrix representation). Component $\tilde{B} \subseteq B$ is the set of buffer channels that affect the timing relation t.

To generate the EMR, we propose an extended symbolic simulation during which, tokens are assigned with vectors $\boldsymbol{v} = [\ (t_1, \tilde{B}_1) \quad \cdots \quad (t_m, \tilde{B}_m)\ ]$, where t_n are the time-stamps of the token and $\tilde{B}_n$ are the sets of buffer channels that affect their respective time-stamps t_n. We refer to $\boldsymbol{v}$ as the *Extended Symbolic Time-stamp Vector (ESTV)*. To improve readability, we refer to t_n and $\tilde{B}_n$ of the ESTV of a token τ, as t_n and $\tilde{B}_n$ of τ. We introduce functions $t_n(\tau)$ and $\tilde{B}_n(\tau)$ that map token τ to t_n and $\tilde{B}_n$ of its ESTV. We compute these maps for every token during the simulation.

We illustrate the proposed simulation on the example SDF scenario. Figure 6 shows the simulation steps. Figure 6a shows the initial distribution of the tokens and Figures 6b-6f respectively show the distribution of tokens after a firing according to the sequence $\sigma = PQRQR$ of actor firings. Figure 6a shows the initial distribution of the tokens and their ESTVs. Since the tokens are initial, their time-stamps are not affected by any buffer and therefore, the buffer sets of all ESTVs are empty. The time-stamp components of the ESTVs are not discussed since they are exactly the same as in the traditional symbolic simulation (see Section 3.3).

In Figure 6b, P has consumed δ_1 and δ_2, fired, and produced τ_1 and τ_2. To compute $\tilde{B}_n(\tau_1)$, we first need to find out, which of the consumed tokens influence the time-stamp $t_n(\tau_1)$ and determine its value. According to actor firing semantics, consumed tokens with the latest time-stamp determine the time-stamp of the produced token. For example, $t_1(\tau_1)$ is determined by $t_1(\delta_1)$, since $t_1(\delta_1) = 0$ is later than $t_1(\delta_2) = -\infty$. We say that δ_1 is the dominating token for $t_1(\tau_1)$. Since $t_1(\tau_1)$ is dominated by $t_1(\delta_1)$ and, δ_1 is on the capacity channel of a, we can conclude that $t_1(\tau_1)$ is influenced by a. Therefore, $\tilde{B}_1(\tau_1) = \{a\}$. For a similar reason, $t_2(\tau_1)$ is dominated by δ_2 and therefore, $\tilde{B}_2(\tau_1) = \{a\}$. Since $t_3(\tau_1) = -\infty$, this time-stamp is affected with nothing and $\tilde{B}_3(\tau_1)$ is empty. Since τ_2 is produced by the same firing that produced τ_1, $\boldsymbol{v}_{\tau_2} = \boldsymbol{v}_{\tau_1}$.

In Figure 6c, Q has consumed τ_1 and δ_3, fired, and produced τ_3 and the final token δ_1 (not to be confused with the initial token δ_1). For $t_1(\delta_1)$ and $t_2(\delta_1)$, τ_1 is the dominating token because, $t_1(\tau_1) = 2 > t_1(\delta_3) = -\infty$ and $t_2(\tau_1) = 2 > t_2(\delta_3) = -\infty$. Since $t_1(\delta_1)$ is dominated by τ_1, it is influenced by all buffers that influence $t_1(\tau_1)$. This means, the buffers that belong to $\tilde{B}_1(\tau_1)$, should also belong to $\tilde{B}_1(\delta_1)$. For a similar reason, the buffers that belong to $\tilde{B}_2(\tau_1)$ also belong to $\tilde{B}_2(\delta_2)$. Token δ_3 is the dominating token for $t_3(\delta_1)$. Since δ_3 is on the capacity channel of buffer b, $\tilde{B}_3(\delta_1) = \{b\}$. Figures 6d-6f show the rest of the simulation steps. Figure 6f shows the ESTVs of the tokens remaining on the channels after the scenario is completely executed. By vertically augmenting these vectors we obtain the extended matrix $\mathbf{G}^e$.

We formalize our symbolic simulation by defining the ESTV of a token ρ that is produced by a symbolic firing of an actor a with execution time $e(a)$, consuming a set T of tokens. Similar to the traditional symbolic firing, $t_n(\rho)$, is defined as follows.

$$t_n(\rho) = \max_{\tau \in T}\{t_n(\tau)\} + e(a) \tag{4}$$

ALGORITHM 1: Compute the EMR of an SDF scenario

Input: An SDF scenario F with I initial tokens
Output: EMR $\mathbf{G}^e$

1 $Y = \{(\delta, \text{ESTV}(\delta)) \mid \delta \in \mathit{InitialTokens}\}$;
2 σ = computeSeqSchedule(F);
3 **for** $j = 1$ *to* length(σ) **do**
4 $\quad a$ = getActor(σ, j);
5 $\quad$ Consume tokens $T \subseteq Y$ and fire a;
6 $\quad Y = Y \setminus T$;
7 $\quad$ **for** $n = 1$ *to* I **do**
8 $\quad\quad t_n = \max_{\tau \in T}\{t_n(\tau)\} + e(a)$;
9 $\quad\quad T_n = \{\tau \in T \mid t_n(\tau) + e(a) = t_n\}$;
10 $\quad\quad \tilde{B}_n = \{b \in B \mid \exists_{\tau \in T_n}[b \in \tilde{B}_n(\tau) \vee bf(\tau, b)]\}$;
11 $\quad$ **end**
12 $\quad \boldsymbol{v} = [\ (t_1, \tilde{B}_1) \quad \cdots \quad (t_I, \tilde{B}_I)\ ]$;
13 $\quad V = \{(\rho, \boldsymbol{v}) \mid \rho \in \mathit{ProducedTokens}\}$;
14 $\quad Y = Y \cup V$;
15 **end**
16 $\mathbf{G}^e = [\boldsymbol{v}(\delta)]$ for all $\delta \in Y$;

To define $\tilde{B}_n(\rho)$, we first need to define the dominating tokens for $t_n(\rho)$. The dominating tokens for $t_n(\rho)$ are all consumed tokens with the latest t_n. Therefore, t_n of the dominating tokens determines $t_n(\tau)$. In other words, if τ is a dominating token for $t_n(\rho)$, then $t_n(\tau) + e(a) = t_n(\rho)$. The set $T_n(\rho)$ of the dominating tokens for $t_n(\rho)$ is defined as follows.

$$T_n(\rho) = \{\tau \in T \mid t_n(\tau) + e(a) = t_n(\rho)\} \tag{5}$$

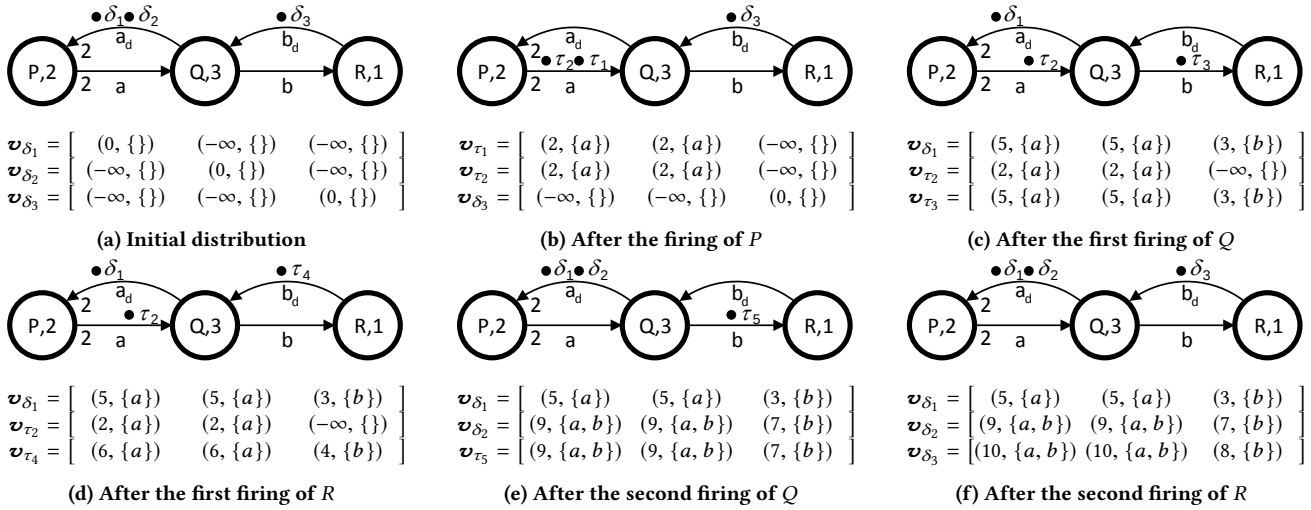
A buffer b belongs to $\tilde{B}_n(\rho)$ if at least one of the following two conditions hold. First, if a token that belongs to T_n is on the capacity channel of b. Second, if b belongs to $\tilde{B}_n(\tau)$ for any $\tau \in T_n$. Let $bf(\tau, b)$ be a function that returns *true* if token τ is on the capacity channel of buffer b and *false* otherwise. The function $\tilde{B}_n(\rho)$ is defined as follows.

$$\tilde{B}_n(\rho) = \left\{b \in B \mid \exists_{\tau \in T_n}[b \in \tilde{B}_n(\tau) \vee bf(\tau, b)]\right\} \tag{6}$$

Algorithm 1 sketches the proposed symbolic simulation method to obtain the EMR of a deadlock-free SDF scenario. Line 1 assigns ESTVs to all initial tokens and stores them in a set Y. In Line 2, a deadlock free actor firing sequence that complies with repetition vector of the scenario is generated and stored in σ (such a schedule exists as the given scenario is deadlock-free). Starting from the first firing in σ, for every firing in the sequence the following actions are performed. The actor responsible for the firing is recognized. The actor consumes a set T of tokens from Y and fires. The consumed tokens are removed from Y (lines 4-6). The ESTVs of the consumed tokens and the execution time of the actor are used in lines 8-10 to compute the elements of the ESTV $\boldsymbol{v}$ according to equations Eqs. 4 and 6. $\boldsymbol{v}$ is constructed from its elements in line 12. In line 13, the tokens are produced and assigned with $\boldsymbol{v}$. These tokens are added to Y in line 14. Finally, the ESTVs of the tokens in Y are vertically augmented to generate the EMR of the scenario (line 16).

6 THE CRITICAL BUFFERS OF AN FSM-SADF

This section defines the critical buffers of an F_d. We first provide a definition for the case that at least one of the scenarios in F_d



(a) Initial distribution **(b) After the firing of P** **(c) After the first firing of Q**

(d) After the first firing of R **(e) After the second firing of Q** **(f) After the second firing of R**

Figure 6: Token distributions in the symbolic simulation of the example SDF scenario

deadlocks due to insufficient storage space (see Section 3.2). Then we provide a definition for an F_d with a deadlock-free execution.

For the deadlock case we use a definition from Stuijk et al. [15]. For every scenario that deadlocks, a Deadlock Dependency Graph (DDG) is defined. The DDG (D, E) is generated from the scenario graph when it is in the deadlock state. The set D of nodes contains a node for every actor in the scenario graph. The set E of edges contains an edge c from an actor P to actor Q if and only if firing of P is prohibited by a lack of tokens on a channel c from Q to P. The definition of critical buffers for the deadlock case is as follows.

Definition 6.1. A buffer channel b in the scenario graph of a deadlock scenario is critical if the capacity channel $\overleftarrow{b}$ is in a cycle of the DDG of the scenario. A common buffer u of an FSM-SADF is critical if $b = b_s(u)$ is critical for some deadlock scenario s.

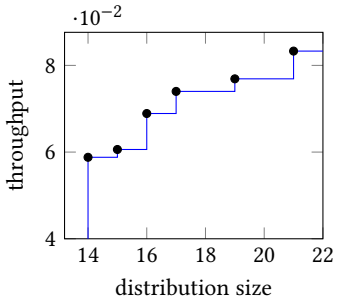
For the deadlock-free F_d, we can generate the extended (max, +) automaton (an example is shown in Figure 5). The EMPA is obtained by replacing the matrices $\mathbf{G}(s)$ with $\mathbf{G}^e(s)$ in the MPA generation method of Geilen et al. [5] (see Section 3.6). A critical cycle of the EMPA is a cycle with the maximum cycle ratio. The definition of critical buffers for the deadlock-free case is as follows.

Definition 6.2. A buffer channel b in the scenario graph of a scenario in a deadlock-free F_d is critical if it is in a critical cycle of the EMPA of F_d. A common buffer u of an FSM-SADF is critical if $b = b_s(u)$ is critical for some scenario s.

In the rest of the paper, the critical buffers refer to either the deadlock case or the deadlock-free case, as clear from the context.

7 DESIGN SPACE EXPLORATION

Using the definitions in Section 6, we can find the trade-offs between the distribution size and the throughput, i.e., the Pareto space. Figure 7 shows the Pareto space obtained for the FSM-SADF in Figure 3. The interesting points on the throughput-distribution size trade-off space are the points that have the minimum distribution size for a given throughput, i.e., the Pareto points.



Thr	dist	seq
0.058	$\langle 6, 8\rangle$	s_1^ω
0.060	$\langle 7, 8\rangle$	s_1^ω
0.068	$\langle 8, 8\rangle$	$(s_1 s_2)^\omega$
0.074	$\langle 9, 8\rangle$	$(s_1 s_2)^\omega$
0.076	$\langle 9, 10\rangle$	$(s_1 s_2)^\omega$
0.083	$\langle 9, 12\rangle$	$(s_1 s_2)^\omega$

Figure 7: Throughput-buffering trade-offs for Figure 3

Formally, for an FSM-SADF, a storage distribution d with throughput *Thr* is minimal if and only if there is no storage distribution d' with throughput *Thr*$'$ such that $|d'| \leq |d|$ and *Thr*$' >$ *Thr* or $|d'| < |d|$ and *Thr*$' \geq$ *Thr*. The table next to the figure, lists all Pareto points from the lowest positive throughput to the maximum achievable throughput. The table shows for every Pareto point also the capacity of the buffers u_a and u_b and the scenario sequence that limits the throughput. For all distributions for which the FSM-SADF deadlocks the throughput is 0. Therefore the zero distribution $\langle 0, 0\rangle$ is the minimal distribution for zero throughput. Obviously, zero throughput is not an interesting design point, thus we do not show it in Figure 7. Observe that the smallest distribution with a throughput larger than zero is $6 + 8 = 14$ tokens, as scenario s_1 deadlocks with any capacity lower than 6 tokens for the buffer a and, scenario s_2 deadlocks with any capacity lower than 8 tokens for the buffer b. The sequence $(s_1 s_2)^\omega$ limits the maximum achievable throughput to $1/12$. The maximal throughput can be achieved with the minimum distribution size of $9 + 12 = 21$ as shown in the figure.

Algorithm 2 sketches the adapted DSE. The exploration is based on the monotonicity property of FSM-SADF [5]. Monotonicity ensures that an increase in the capacity of a buffer cannot lower the throughput. The input is an FSM-SADF F and the output is the set P

ALGORITHM 2: Find all minimal storage distributions

Input: An FSM-SADF F
Output: The set P of all pairs (d, Thr) s.t. d are minimal

1 Thr_{max} = computeThrMax(F);
2 $I = \{\langle 0, \cdots, 0\rangle\}$; $P = \{\}$;
3 **repeat**
4 d = getSmallestDist(I); $I = I \setminus \{d\}$;
5 F_d = createFSMSADF(F, d);
6 **if** F_d *is deadlock-free* **then**
7 M_d = computeEMPA(F_d);
8 (Thr, C) = computeThrAndACriticalCycle(M_d);
9 **else**
10 G_d = computeDDG(F_d);
11 $Thr = 0$; C = findACycle(G_d);
12 **end**
13 $P = P \cup \{(d, Thr)\}$;
14 U = getCriticalBuffers(C);
15 **foreach** $u \in U$ **do**
16 d' = copyDist(d);
17 $d'(u) = d(u) + \text{step}(u)$;
18 $I = I \cup \{d'\}$;
19 **end**
20 **until** $\exists_{(d,Thr)\in P}[Thr = Thr_{max} \wedge \nexists_{d'\in I}[|d'| = |d|]]$;
21 P = removeNonOptimalDists(P);

of all pairs (d, Thr) such that d is a minimal distributions and Thr is its throughput. In the first line, the maximal throughput Thr_{max} is computed by removing all capacity channels from scenario graphs and using the throughput analysis of Geilen et al. [6] (we assume that the modelled application has a bounded throughput, otherwise, its trade-off space is not finite). The set I of the storage distributions that will be explored by the algorithm is initialized with zero distribution $\langle 0, \ldots, 0\rangle$ and, P is initially empty (line 2). Lines 4-19 are repeated until all minimal distributions with Thr_{max} are in P. A distribution d with the smallest size is picked and then removed from I (line 4). F_d is created from d and F in line 5. If F_d is deadlock-free, the EMPA of the F_d is created in line 7. From the EMPA, the throughput and a critical cycle are obtained and stored in Thr and C, respectively (line 8). If F_d deadlocks, then the algorithm generates the DDG of F_d, finds a cycle C in the DDG and $Thr = 0$ (lines 10-11). The pair (Thr, d) is added to P in line 13. A set U of critical buffers are obtained from C in line 14 using the definitions provided in Section 6. For every $u \in U$, a new distribution d' is generated from d by increasing the capacity of u by one step, then it is added to the distributions pool I (lines 16-18). The step size for a buffer is the minimum number of tokens that if added to the capacity of the buffer, it may break the immediate actor firing dependencies on that buffer. We define the step size of a common buffer as the minimum of the input and output rates of the corresponding buffer channel over all scenarios. In Figure 3, the step size for u_a and u_b are $\min\{3, 4, 1, 2\} = 1$ and $\min\{2, 6, 4, 6\} = 2$, respectively. In the final step (line 21), all the non-minimal distributions are removed from P, after which it is returned. To prove the correctness of the algorithm, we proceed as follows (adapted from Stuijk et al. [15]).

Lemma 7.1. *Given a storage distribution d_i with throughput $Thr_i > 0$, for any storage distribution $d_j \preceq d_i$ for which F_{d_j} deadlocks in some scenario s, in any cycle in the DDG of s, there exists a capacity channel $\overleftarrow{b} = c_s(u)$ for which, $d_i(u) > d_j(u)$.*

Proof. Since $Thr_i > 0$, F_{d_i} is deadlock-free. Since F_{d_i} is deadlock-free, but F_{d_j} deadlocks in scenario s, d_i has resolved any cycle in DDG of scenario s in F_{d_j}. To resolve a cycle in the DDG of F_{d_j}, the capacity of at least one buffer channel that its corresponding capacity channel is on the cycle, must be increased. For every capacity channel $\overleftarrow{b} = c_s(u)$ that the capacity of its corresponding buffer must be increased, we have $d_i(u) > d_j(u)$. □

Lemma 7.2. *Given a storage distribution d_i with throughput Thr_i, for any storage distribution $d_j \preceq d_i$ with throughput $0 < Thr_j < Thr_i$, in any critical cycle in the EMPA of F_{d_j} there exists a buffer channel $b = b_s(u)$ for which, $d_i(u) > d_j(u)$.*

Proof. Since d_j has a positive throughput, it is deadlock free and F_{d_j} has an EMPA. Since F_{d_i} has a higher throughput than F_{d_j}, we can conclude that any critical cycle in the EMPA of F_{d_j} has been resolved in the EMPA of F_{d_i}. To resolve a critical cycle in the EMPA of F_{d_j} the capacity of at least one buffer channel on the critical cycle must be increased. Note that any critical cycle in the EMPA of F_{d_j} contains at least one edge with a non-empty set of buffers channels (otherwise Thr_j would be maximal, contradicting $Thr_j < Thr_i$). For every buffer channel $b = b_s(u)$ that its capacity must be increased, we have $d_i(u) > d_j(u)$. □

Lemma 7.3. *Given a storage distribution d_i with throughput Thr_i and a storage distribution d_j such that $d_j \preceq d_i$ with throughput $Thr_j < Thr_i$. From d_j, Algorithm 2 explores a storage distribution d_k for which $d_j \preceq d_k \preceq d_i$, $\left|d_j\right| < |d_k|$ and $Thr_j \leq Thr_k \leq Thr_i$.*

Proof. If F_{d_j} deadlocks, according to Lemma 7.1 and, if it is deadlock-free, according to Lemma 7.2, there exists a buffer u in the set U of critical buffers such that $d_i(u) > d_j(u)$. Therefore, the capacity of u in d_j can be increased before the capacity of u becomes equal to the capacity of u in d_i. Since u is critical, Algorithm 2 increases the capacity of u which results in a new distribution d_k. As the capacity of u is increased but not beyond its capacity in d_i, it holds that $d_j \preceq d_k \preceq d_i$ and $\left|d_j\right| < |d_k|$. From the monotonicity property it holds that $Thr_j \leq Thr_k \leq Thr_i$. □

Theorem 7.4. *Algorithm 2 obtains all minimal storage distributions in P.*

Proof. Let d_i be a minimal storage distribution with throughput Thr_i. We show that the algorithm explores d_i starting from any distribution $d_j \preceq d_i$ already explored by the algorithm. We show by induction that d_i is explored from any d_j, such that $|d_i| - \left|d_j\right| = n$ for any non-negative integer n. The base case, $n = 0$, is trivial, because $|d_i| = \left|d_j\right|$ and $d_j \preceq d_i$ means that $d_i = d_j$, i.e., d_i is explored. Now let's assume d_i is explored from any explored d_j such that $|d_i| - \left|d_j\right| = k$ and $0 \leq k \leq n$. We need to show that the algorithm also explores d_i from any explored distribution d_j such that $|d_i| - \left|d_j\right| = n+1$. Lemma 7.3 states that from a distribution d_j, a distribution $d_k \succeq d_j$ is explored such that $|d_k| > \left|d_j\right|$. According to the algorithm, d_k is generated from d_j by increasing the capacity of at least one buffer by a step of some size s. Therefore $|d_k| - \left|d_j\right| = s$. By comparing $|d_i|$ and $|d_k|$ we conclude that $|d_i| - |d_k| = n + 1 - s$.

Table 1: Experimental results on FSM-SADF models

	WLAN	MP3 Decoder	SUSAN	Figure 3	H.263 Decoder	Modem	Sample Rate	Satellite
nrBuffers	8	46	4	2	3	35	10	48
nrParetoPoints	2	3	7	6	36	3	3	2
nrMinDist	2	3	7	6	146	3	3	2
nrVisitedDist	3	32	12	10	193	7	5	6
maxThr($\times 10^{-4}$)	2.5	0.0017	0.066	830	0.03	620	10	7.5
distSizeMaxThr	10	179	20	21	1224	72	44	1586
minPosThr($\times 10^{-4}$)	2.4	0.0016	0.012	5.80	0.015	320	9.2	9.4
distSizeMinThr	8	172	8	14	1189	70	42	1588
RunTime Alg. 2	3ms	4559ms	11ms	19ms	1296s	74ms	545ms	419s
RunTime [15]	n/a	n/a	n/a	n/a	349ms	1ms	8ms	514ms

Since $s \geq 1$, $n + 1 - s \leq n$ and, according to induction hypothesis, d_k is explored by the algorithm. Since d_i is explored from d_k, and d_k is explored from d_j, we conclude that d_i is explored from d_j. The algorithm starts from the zero distribution. Since $\langle 0, \cdots, 0 \rangle \preceq d_i$ for any d_i, the algorithm explores any minimal distribution d_i. □

8 EVALUATION

We implemented Algorithm 1 and Algorithm 2 in the SDF3 tool [14]. We applied Algorithm 2 to FSM-SADF models of several realistic applications. The set of FSM-SADF application models contains an MP3 decoder [5], an edge detection algorithm known as SUSAN [16] and a WLAN [5]. We also transformed a set of SDF application models to FSM-SADF (the FSM-SADF contains a single scenario that is defined as the set of actor firings in one iteration of the SDF graph) to apply our analysis and compare the results. The set of SDF applications include an H.263 decoder [15], a Modem [1], a sample rate converter [1] and a satellite receiver [13].

The buffer sizing results are shown in Table 1. For every model, we report the number of sized buffers, Pareto points, minimum distributions, visited storage distributions, the maximum and minimum positive throughput and the size of minimal distributions for which the throughput numbers are obtained. We have excluded the trivial zero storage distribution from the number of Pareto points, minimum distributions and visited storage distributions, as it is not an interesting design point. Note that the number of minimal distributions might be more than the number of Pareto points, as there might be distributions with the same size but different configurations in a Pareto point. The run-time of Algorithm 2 is reported for all models. The run-times include all function calls in the algorithm including the run-time of Algorithm 1, as it is called by Algorithm 2. For SDF graphs, we compared the results of our buffer sizing algorithm with the SDF buffer sizing method provided by Stuijk et al. [15]. We observed that the results are identical. We also report the run-time of their algorithm. All run-times are measured on an Ubuntu server with a 3.8Ghz processor.

Observe that the run-time of Algorithm 2 is affected by the number of parameters. First, the input model should have a bounded throughput, otherwise, its trade-off space is not finite and the algorithm never terminates. If the model deadlocks even with unbounded buffers, then the model is incorrect. This case can be detected at the first step of the algorithm. For all other cases the algorithm returns all optimal distributions. The complexity analysis for those cases is as follows.

To compute the throughput of a distribution we first need to compute the EMR of all scenarios. The time complexity of this step is $O(|S| \cdot |I|^2 \cdot |\sigma|)$ where $|S|$, $|I|$ and $|\sigma|$ are the number of scenarios, initial tokens and actor firings respectively. Since the distributions are modelled by adding initial tokens on the capacity channels, computing the EMRs scales quadratically in the distribution size (the total number of added tokens equals the size of the distribution).

The time complexity of an MCR analysis and obtaining a critical cycle (Karp [10]) on a MPA is $O(|R| \cdot |E|)$, where $|R|$ and $|E|$ are the number of nodes and edges of the MPA. The MPA of an F_d by definition has a number of nodes $|R| = |W| \times |I|$ and edges $|E| = |S| \times |I|^2$, where $|W|$ is the number of FSM states. Consequently, MCR analysis has a cubic time complexity in the size of the distribution. Therefore computing the throughput of distribution has a worst-case cubic complexity in the size of the distribution. The results confirm that the analysis time is longer for the models with larger distributions compared to the models with smaller distributions.

9 CONCLUSION

When designing multi-media and communication applications on multi-processor systems, there is a trade-off between the amount of allocated storage space to the application and the throughput of the application. A complicated design problem is to find the minimal capacity requirements for buffers to execute an application under a given throughput constraint. To formalize the problem, a suitable model of computation is needed to mathematically describe the application. Nowadays, applications are dynamic, meaning that their behaviour changes at run-time. FSM-SADF is a suitable model for dynamic applications. In this paper we provided a technique to find all optimal throughput-storage space trade-offs for applications using FSM-SADF models. The analysis performs a guided design space exploration that cuts-off the exploration space during the exploration without losing any optimal points and consequently, terminates in a reasonably short time. We showed that our method is practical by applying it to a number of realistic application models.

A LIST OF ABBREVIATIONS

Table 2: Abbreviations

Abbreviation	Explanation
CSDF	Cyclo-Static Dataflow
DDG	Deadlock Dependency Graph
DSE	Design Space Exploration
EMPA	Extended (max, +) Automaton
EMR	Extended Matrix Representation
ESTV	Extended Symbolic Time-stamp Vector
FSM	Finite State Machine
FSM-SADF	Finite-State-Machine Scenario-Aware Dataflow
FTs	Final Tokens
IDCT	Inverse Discrete Cosine Transformation
IQ	Inverse Quantization
ITs	Initial Tokens
MC	Motion Compensation
MCR	Maximum Cycle Ratio
MoC	Model of Computation
MPA	(max, +) Automaton
RC	Reconstruction
SDF	Synchronous Dataflow
SDFG	Synchronous Dataflow Graph
VLD	Variable Length Decoding

ACKNOWLEDGMENTS

This research is supported by the ARTEMIS joint undertaking through the ALMARVI project (621439) and ITEA 3 Project 14014 ASSUME.

REFERENCES

[1] S. Bhattacharyya, P. Murthy, and E. 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. https://doi.org/10.1023/A:1008052406396

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

[3] B. Bodin, A. Munier-Kordon, and B. de Dinechin. 2013. Periodic schedules for Cyclo-Static Dataflow. In *Proc. 11th IEEE Symposium on Embedded Systems for Real-time Multimedia*. 105–114. https://doi.org/10.1109/ESTIMedia.2013.6704509

[4] K. Denolf, A. Chirila-Rus, P. Schumacher, R. Turney, K. Vissers, D. Verkest, and H. Corporaal. 2007. A Systematic Approach to Design Low-power Video Codec Cores. *EURASIP J. Embedded Syst.* 1 (2007), 42–42.

[5] M. Geilen, J. Falk, C. Haubelt, T. Basten, B. Theelen, and S. Stuijk. 2017. Performance Analysis of Weakly-Consistent Scenario-Aware Dataflow Graphs. *Signal Processing Systems* 87, 1 (2017), 157–175. https://doi.org/10.1007/s11265-016-1193-7

[6] M. Geilen and S. Stuijk. 2010. Worst-case Performance Analysis of Synchronous Dataflow Scenarios. In *Proc. 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES/ISSS '10)*. ACM, New York, NY, USA, 125–134. https://doi.org/10.1145/1878961.1878985

[7] R. Govindarajan, Guang R. Gao, and Palash Desai. 2002. Minimizing Buffer Requirements under Rate-Optimal Schedule in Regular Dataflow Networks. *Journal of VLSI signal processing systems for signal, image and video technology* 31, 3 (01 Jul 2002), 207–229. https://doi.org/10.1023/A:1015452903532

[8] C. T. Hwang, J. H. Lee, and Y. C. Hsu. 1991. A formal approach to the scheduling problem in high level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 10, 4 (Apr 1991), 464–475. https://doi.org/10.1109/43.75629

[9] A. Jerraya and W. Wolf. 2004. *Multiprocessor systems-on-chips*. Elsevier.

[10] R. Karp. 1978. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics* 23, 3 (1978), 309 – 311. https://doi.org/10.1016/0012-365X(78)90011-0

[11] E. Lee and D. Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (Sept 1987), 1235–1245. https://doi.org/10.1109/PROC.1987.13876

[12] P. Murthy. 1996. *Scheduling Techniques for Synchronous and Multidimensional Synchronous Dataflow*. Ph.D. Dissertation. EECS Department, University of California, Berkeley.

[13] S. Ritz, M. Willems, and H. Meyr. 1995. Scheduling for optimum data memory compaction in block diagram oriented software synthesis. In *Proc. 1995 International Conference on Acoustics, Speech, and Signal Processing*, Vol. 4. 2651–2654 vol.4. https://doi.org/10.1109/ICASSP.1995.480106

[14] S. Stuijk, M. Geilen, and T. Basten. 2006. SDF3: SDF for free. In *Proc. 6th International Conference on Application of Concurrency to System Design*. IEEE, 276–278.

[15] 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. https://doi.org/10.1109/TC.2008.58

[16] R. van Kampenhout, S. Stuijk, and K. Goossens. 2017. Programming and analysing scenario-aware dataflow on a multi-processor platform. In *Proc. Design, Automation Test in Europe Conference Exhibition (DATE), 2017*. 876–881. https://doi.org/10.23919/DATE.2017.7927110

[17] M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit. 2007. Efficient Computation of Buffer Capacities for Cyclo-Static Dataflow Graphs. In *Proc. 44th ACM/IEEE Design Automation Conference*. 658–663.

[18] P. S. Wilmanns, S. J. Geuns, J. P. H. M. Hausmans, and M. J. G. Bekooij. 2015. Buffer Sizing to Reduce Interference and Increase Throughput of Real-Time Stream Processing Applications. In *Proc. 18th International Symposium on Real-Time Distributed Computing*. 9–18. https://doi.org/10.1109/ISORC.2015.14