Worst-case Performance Analysis of Synchronous Dataflow Scenarios

Marc Geilen
Eindhoven University of Technology
P.O. Box 513
Den Dolech 2
Eindhoven, The Netherlands
m.c.w.geilen@tue.nl

Sander Stuijk
Eindhoven University of Technology
P.O. Box 513
Den Dolech 2
Eindhoven, The Netherlands
s.stuijk@tue.nl

ABSTRACT

Synchronous Dataflow (SDF) is a powerful analysis tool for regular, cyclic, parallel task graphs. The behaviour of SDF graphs however is static and therefore not always able to accurately capture the behaviour of modern, dynamic dataflow applications, such as embedded multimedia codecs. An approach to tackle this limitation is by means of scenarios. In this paper we introduce a technique and a tool to automatically analyse a scenario-aware dataflow model for its worst-case performance. A system is specified as a collection of SDF graphs representing individual scenarios of behaviour and a finite state machine that specifies the possible orders of scenario occurrences. This combination accurately captures more dynamic applications and this way provides tighter results than an existing analysis based on a conservative static dataflow model, which is too pessimistic, while looking only at the 'worst-case' individual scenario, without considering scenario transitions, can be too optimistic. We introduce a formal semantics of the model, in terms of $(\max, +)$ linear system-theory and in particular $(\max, +)$ automata. Leveraging existing results and algorithms from this domain, we give throughput analysis and state space generation algorithms for worst-case performance analysis. The method is implemented in a tool and the effectiveness of the approach is experimentally evaluated.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling Techniques; F.1.1 [**Computation by Abstract Devices**]: Models of Computation

General Terms

Algorithms, Performance, Theory, Verification

Keywords

Synchronous Data Flow, worst-case performance analysis, $(\max, +)$ algebra

1. INTRODUCTION

A number of model-based design approaches [12, 16, 22, 19] for firm real-time parallel streaming applications are based on conservative dataflow actor models such as Synchronous Dataflow [17, 23] or a slight generalisation, Cyclo-Static Dataflow CSDF [2]. Repetitive actors on suitably arbitrated resources can be modelled accurately and with performance guarantees, as data-driven, self-timed dataflow actors. Such actors start their execution as soon as all input data has arrived, take a given time to execute and after that produce their output data. As long as such a dataflow model is analysable, then performance guarantees can be derived at design-time. SDF and CSDF are such analysable models, while certain more expressive dataflow models such as Dynamic Dataflow [4] or Kahn Process Networks [13] are known to be undecidable and can therefore not be used directly for such purpose. SDF and CSDF restrict their actors to produce and consume data with fixed rates per firing or, in case of CSDF, with fixed periodic patterns.

In many modern streaming applications, such as audio or video codecs with advanced data compression schemes, the behaviour of an encoder or decoder can show much more dynamism than can be effectively expressed with an SDF or CSDF graph. A simple example is an MPEG-1 Layer 3 audio decoder, commonly known as MP3. It divides a stereo audio stream into frames of 26ms and may employ five different coding schemes for a frame, depending on the audio content. The different modes employ differently sized subbands (short vs. long blocks) or a mixture of both, partly independently for the left and right audio channel. SDF or CSDF cannot capture such dynamic switching between frame types, except by employing an overly pessimistic model which is conservative for any type of frame.

Scenario-based or mode-based design [29, 12, 18, 19] is an approach in which the dynamic behaviour of an application is viewed upon as a collection of different behaviours, called *scenarios* or *modes*, occurring in certain known or unknown patterns, but each of which is by itself fairly static and predictable in performance and resource usage and can be dealt with by traditional methods. Some of the difficulties are moved, however, to predicting what scenarios occur and to dealing with scenario transitions. We also follow the scenario based approach in this paper by modelling our embedded software and platform combination with the Scenario-Aware Dataflow (SADF) [25] generalisation of (C)SDF. SADF characterises every individual scenario or mode by a specific SDF graph, The SDF graph models tasks with constant,

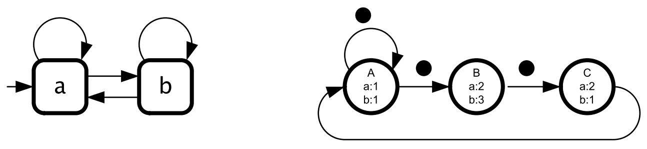


Figure 1: Example SADF graph

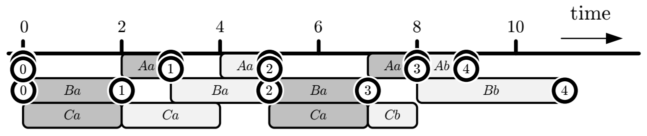


Figure 2: Execution of the example graph for scenarios aaab

worst-case, execution times. Because of the separation in scenarios, the worst-case execution time estimates can be tighter. The possible scenario transitions are captured by a finite state machine (FSM). Figure 3 shows an example of an MPEG video decoder which follows a regular pattern of I and P frames and where the P frames are classified in different scenarios depending on the number of macro blocks in the frame. Figure 4 shows the FSM that specifies the possible scenario sequences. For the MP3 decoder, the frame types can occur in arbitrary order, which can be expressed by a fully connected FSM. In a streaming, distributed embedded software system, the difficulty lies in the fact that different scenarios are concurrently active in different stages of a pipelined implementation. The techniques introduced in [9, 21] and in this paper allow those scenarios to still be handled individually, compositionally. In this paper we introduce a method to derive the tightest possible performance guarantees for the class of SADF graphs. Moreover, it is shown that two simpler approaches do not provide conservative and tight results. Analysing only the worst-case scenario (for instance the one with the lowest throughput) does not provide a guaranteed conservative bound on the behaviour over all scenarios. On the other hand, creating a single SDF graph from the worst-case execution times of actors over all scenarios, can be too pessimistic.

We proceed with a discussion of related work in the following section. Then we discuss technical preliminaries of SDF, SADF and their $(\max, +)$ modelling in Section 3. Section 4 introduces the $(\max, +)$ semantics and state space model for SADF and indicates how throughput and latency can de defined on this state space. An efficient approach for calculating the worst-case throughput of an SADF is given in Section 5. An implementation of the state-space analysis algorithm and its evaluation are discussed in Sections 6 and 7 respectively, followed by concluding remarks and suggestions for future work.

2. RELATED WORK

Throughput analysis of Synchronous Data Flow graphs is studied in [17, 23, 10]. SDF is very fitting for regular streaming applications. The desire to extend the range of applicability to more dynamic models has lead to use of extensions such as Cyclostatic SDF (CSDF) [2], Heterochronous Dataflow (HDF) [11], Kahn Process Networks (KPN) [13, 14], Scenario-Aware Dataflow (SADF) [25] and many others. Some of these such as KPN are not statically analysable and cannot provide hard guarantees. CSDF provides only a limited, strictly periodic kind of dynamic behaviour. HDF is introduced in [11], which does not provide means for performance analysis of the model. Different models fill the spectrum of trade-off between expressiveness, accuracy and analysability. To express more dynamic behaviour, some authors have described their applications as execution in different *modes* or *scenarios* of SDF behaviour [25, 11, 9, 8]. In this paper, we build on the SADF model introduced in [25] in combination with the $(\max, +)$ algebra semantics of [1, 10].

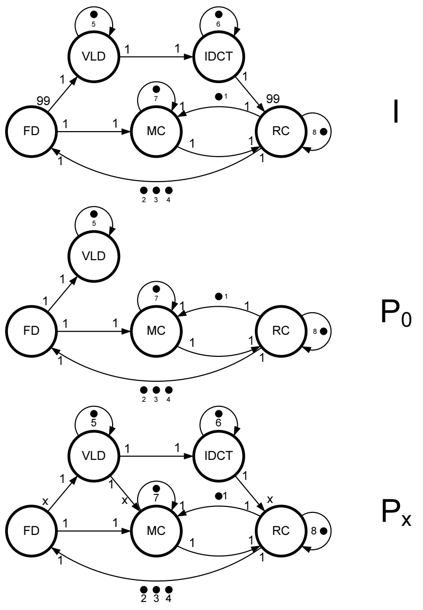


Figure 3: SDF specifications of the MPEG scenarios

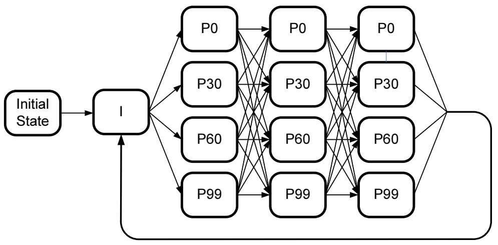


Figure 4: FSM specifying scenario sequences

Existing work on scenarios and modes focusses on sequential software or on other models-of-computation [12, 19]. Real-time calculus also focusses on stream based applications and has mode-based approaches [19], but handling cyclic dependencies is limited to Marked Graphs without modes [26].

The approach of [25] is most closely related to this work. It works with essentially the same model of computation, Scenario-Aware Dataflow Graphs. It introduces an analysis technique that works by building up a global state-space representation of the detailed behaviour of the graph across sequences of scenarios. Transitions are at the level of individual firings of actors. This tends to lead to very large state spaces and tractability issues with larger models. In comparison, the state-space we build in this paper captures a complete iteration of the graph in a particular scenario in a single transition, thereby leading to much smaller state spaces which are easier to handle. Moreover, it is able to handle individual scenario iterations separately, despite the fact that they are pipelined and concurrently active. [25] also deals with a stochastic version of the SADF model. We believe that the method in this paper can be similarly extended to generate a Markov Reward Structure instead of the state space and analyse it for its stochastic performance metrics. [20] also deals with scenarios of SDF behaviour, but in their case only homogeneous SDF graphs are considered (graphs in which all consumption and production rates are equal to one), and only the execution times of a fixed collection of actors can vary with scenarios. In earlier work [9], we have introduced techniques to find linear upper bounds on transient behaviour of an SDF, which also allows the behaviour of an SADF to be analysed, but without exact results, in this work, the exact behaviour is followed and the exact tightest bound on throughput is found. Moreover, in [9], the FSM specification of possible scenario orders could not be taken into account and one would have to assume any scenario order to be possible.

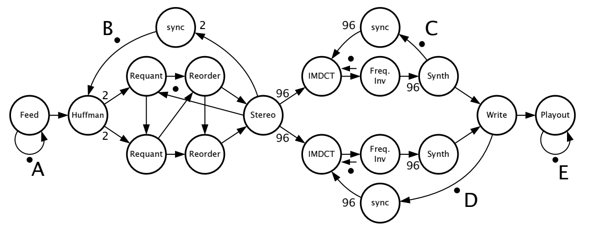
A successful approach to the analysis of timed event graphs, of which SDF graphs are a special case, is linear system theory over the $(\max, +)$ semiring [5, 1]. $(\max, +)$ automata [6] represent systems with such linear dynamics changing over time with different modes depending on the state of the automaton. We exploit this model in this paper as the $(\max, +)$ linear dynamics captures very well the SDF behaviour, while the different modes capture the scenarios.

SDF graphs can be modelled as a specific class of Petri Nets, (Timed) Weighted Marked Graphs. A more restricted class of Petri Nets, Safe Timed Petri Nets, has been analysed using $(\max, +)$ automata in [7]. They model every individual firing, instead of complete iterations and do not explicitly deal with behavioural scenarios. Moreover, they first model firings using the Heaps of Pieces model of [28]. Heaps of Pieces are a generalisation of Gantt charts over multiple resource and data dependencies and have a cute analogy to the stacking of staircase shaped blocks as in the Tetris computer game. A Heaps of Pieces model can be translated to a $(\max, +)$ automaton and can so be analysed, for instance for worst-case behaviour. In this paper we do not capture single firings, but instead entire iterations of an SDF graph and the way pieces are defined in the Heaps of Pieces model does not allow the characterisation of complete iteration dependencies. However, a direct translation into a $(\max, +)$ automaton is possible and used in this paper. We then use the analysis results of $(\max, +)$ automata of [6] to derive performance analysis methods for SDF scenarios. Brilman and Vincent have studied the stochastic analyses of such systems [3].

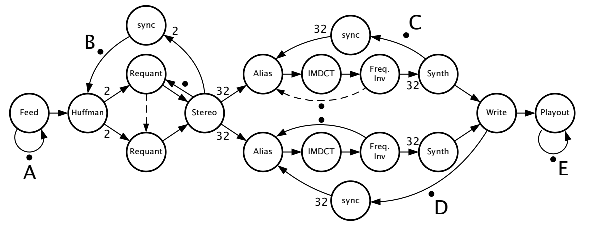
3. PRELIMINARIES

3.1 Synchronous Dataflow Graphs

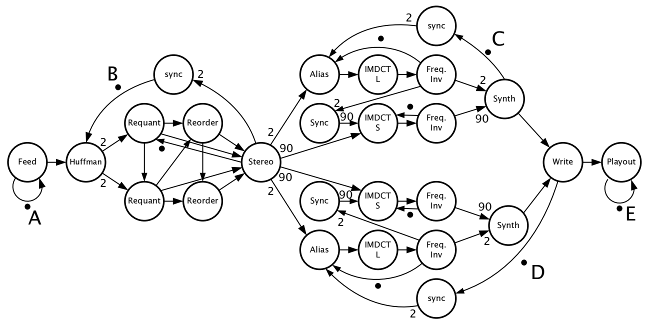
In this paper, we study a class of Scenario-Aware Dataflow Graphs [25]. This model captures behaviour as a dynamic switching between scenarios, each of which are captured by a static Synchronous Dataflow Graph (SDFG) [17, 23]. An SDFG is a directed graph of *actors* and *channels*. An SDFG *models* the behaviour of a *real* dataflow application on a *real* platform. Figure 5 shows a block diagram of an MP3 decoder architecture and its mapping on a platform with three processors. Figure 6(a)-(c) show examples of SDFGs that model three of the five different frame-type scenarios of an MP3 audio decoder as they are mapped on the particular platform. The missing two graphs are merely combinations of different parts of the three shown, when left and right audio channel use different encodings. The synchronisation dependencies between frames arise particularly from the modelled platform mapping. The circles are called actors and represent the individual computations or synchronisation points. Actors can *fire*. When an actor fires, it consumes and produces fixed amounts of *tokens* on the FIFO channels to which it is connected. It can only fire if sufficient tokens are available on the channels from which it consumes. The rates of production and consumption are indicated next to the channel ends. For readability this is omitted in those cases where the rate is equal to 1. Tokens thus capture dependencies between actor firings. Such dependencies may be data dependencies, but also dependencies on shared resources. Actors do not always represent actual computations, but may also be used to model communication, or synchronisation, for instance to express more complicated schedules (such as the Sync actors in Figure 6). For a more detailed, formal, semantics of SDF and its properties we refer to [17, 23, 10].



(a) scenario short blocks left+right



(b) scenario long blocks left+right



(c) scenario mixed blocks left+right

Figure 6: SDF models of two of the MP3 scenarios

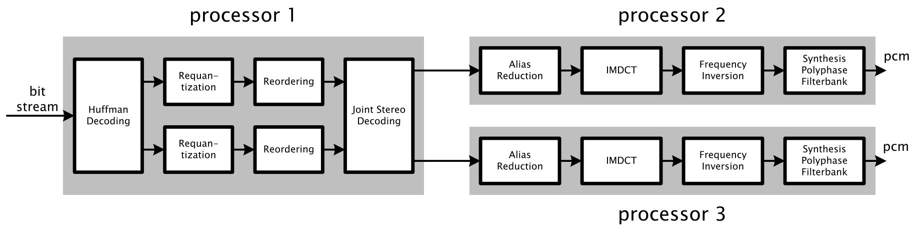


Figure 5: MP3 realisation

To model performance aspects, firings of actors may take time. In typical timed SDF [23], actor firings are modelled with a constant execution time, usually an upper bound on the real execution time of the corresponding computation of the actual application. Note that by modelling scenarios separately we are able to give tighter upper bounds for actors. Because the SDF model of computation is monotone (an earlier or shorter firing of an actor cannot lead to another actor firing occurring later), one can use worst-case execution times in the model and get from the model an upper bound on the actual firing times and data production times of the real application in practice. The so-called *self-timed execution* is a schedule in which every actor firing takes place as soon as possible, immediately when all required tokens are available. In this way, the self-timed execution represents the best (tightest) bound that can be given on the timing behaviour of the application based on the SDF model. That is why the self-timed execution is of special interest.

The constant rates with which tokens are produced and consumed make SDFGs execute in fixed repetitive patterns, called *iterations*. An iteration consists of a collection of actor firings that together have no net effect on the position and numbers of tokens on channels. Moreover, they typically also constitute a coherent collection of computations, for instance processing of a frame in an audio or video stream, such as the audio frames of the MP3 decoder. For the example graph of Figure 1, (in both scenarios) an iteration consist of a single firing of each of the actors. Initially the graph has a specific distribution of its initial tokens across the graph. This distribution remains invariant with individual iterations performed by the graph, independent of the scenario that is executed. The production times of this collection of initial tokens at the end of an iterations exactly captures the starting condition for the next iteration, be it in the same or in a different scenario. This is illustrated in Figure 2, which shows an initial execution of a sequence of four scenarios, *aaab*. The first firing of actor A for example starts at time 2, because it requires an output token from C which needs to complete a firing first. Iterations are alternatingly coloured dark and light grey. Observe that the iterations overlap indicating some degree of pipelining in the execution. The initial tokens are drawn in the row of the actor that produces them and are numbered with the iterations. Observe that the collection of initial tokens produced at the end of an iteration contains enough information to determine the timing of the following iteration. Similarly, for the MP3 scenario graphs, the initial tokens labelled A to E, reoccurring in every scenario graph, carry dependencies between the iterations of MP3 frames. The MP3 scenario graphs have extra initial tokens which do not represent dependencies between iterations. These tokens are assumed to be initially available as early as they are needed. In the remainder of the paper, we assume given a finite set S of scenarios and for every scenario in S a corresponding SDFG with the same number, N, of numbered initial tokens.

3.2 $(\max, +)$-characterisation of SDF Scenarios

We use $(\max, +)$ *algebra* [1] to capture the semantics of our scenarios of SDF graphs. Two essential characteristics of the self-timed execution of an SDFG are *synchronisation*, when the graph waits for sufficient input tokens to fire, and *delay*, when an actor starts firing it takes a fixed amount of time before it completes and produces its output tokens. These two elements correspond well to the $(\max, +)$ operators max and addition. If T is the set of tokens required by an actor to start firing and for every $\tau \in T$, t_τ is the time when the token becomes available, then the starting time of the firing of the actor is given by: $\max_{\tau \in T} t_\tau$. Let further e be the execution time of that actor, then the output tokens produced by that actor become available for consumption by other actors at: $\max_{\tau \in T} t_\tau + e$, which is a $(\max, +)$ expression.

The behaviour of a dataflow graph can be characterised completely by the times at which the tokens in channels are produced, which capture exactly the dependencies between iterations that are crucial to studying scenario transition behaviour. We particularly consider the collection of tokens that exist in their various channels in between iterations, the initial token configuration. The production times of these tokens are collected in a vector $\boldsymbol{\gamma}$ consisting of as many entries as there are initial tokens in the graph (e.g., five (A-E) in Figure 6). After one iteration, by definition [17], there exist the same number of tokens in the same channels, but with different times at which they are produced; $\boldsymbol{\gamma}_0$ becomes after an iteration of the graph a new vector $\boldsymbol{\gamma}_1$ of the same size. We refer to such a vector as a *time-stamp vector*. The process of execution of an SDFG can be captured by means of a matrix-vector multiplication in $(\max, +)$ [1]. In this algebra, addition and max operator take the role of multiplication and addition respectively of traditional linear algebra. The evolution of the SDF graph, characterised by the matrix $\mathbf{G}$, is then governed by the following equation $\boldsymbol{\gamma}_{k+1} = \mathbf{G}\boldsymbol{\gamma}_k$. In an SADF graph, we have such a matrix $\mathbf{G}_s$ for every scenario $s \in S$. If iteration k is executed in scenario s, then $\boldsymbol{\gamma}_{k+1} = \mathbf{G}_s\boldsymbol{\gamma}_k$.

For Figure 1, for instance, we have the following matrices.

$$\mathbf{G}_a = \begin{bmatrix} 1 & -\infty & 3 \\ 1 & -\infty & 3 \\ -\infty & 2 & -\infty \end{bmatrix}, \mathbf{G}_b = \begin{bmatrix} 1 & -\infty & 2 \\ 1 & -\infty & 2 \\ -\infty & 3 & -\infty \end{bmatrix}$$

An entry t at column k and row m in the matrix specifies that there is a minimum distance of t between time-stamps of token k of the previous iteration to token m of the new iteration, following from dependencies in the graph. An entry $-\infty$ signifies that there is no dependency relation. If the last iteration produced its tokens according to the vector $[3;\ 3;\ 2]^T$ (as the tokens labelled '1' in Figure 2) and we execute an iteration in scenario a then the new tokens are produced according to the vector:

$$\boldsymbol{\gamma} = \begin{bmatrix} 1 & -\infty & 3 \\ 1 & -\infty & 3 \\ -\infty & 2 & -\infty \end{bmatrix} \begin{bmatrix} 3 \\ 3 \\ 2 \end{bmatrix} =$$

$$\begin{bmatrix} \max\{1+3,\ -\infty+3,\ 3+2\} \\ \max\{1+3,\ -\infty+3,\ 3+2\} \\ \max\{-\infty+3,\ 2+3,\ -\infty+2\} \end{bmatrix} = \begin{bmatrix} 5 \\ 5 \\ 5 \end{bmatrix}$$

The new vector represent the tokens in Figure 2 labelled '2'.

The Heaps of Pieces model [28] is used in literature to study the behaviour of discrete event systems and in particular Safe Timed Petri Nets (see for instance [7]). For readers familiar with the concept of Heaps of Pieces it is good to observe that Pieces cannot accurately capture a complete iteration of an SDF graph as a single Piece, since Pieces have fixed relative starting times (the lower contour) and fixed completion times (the upper contour). While an iteration is more 'flexible' than a rigid Piece as it consists of a collection of independent actor firings, a short stack of Pieces. However, a single iteration can be accurately captured by a single $(\max, +)$ matrix multiplication.

We briefly introduce some notation related to $(\max, +)$ algebra (see [1] for background on $(\max, +)$ algebra, linear system theory of the $(\max, +)$ semiring). $(\max, +)$ algebra defines the operations of the maximum of numbers and addition over the set $\mathbb{R}^{-\infty} = \mathbb{R} \cup \{-\infty\}$, the real numbers extended with $-\infty$. For readability, we use the standard notation for the max and addition operations instead of the $\oplus$ and $\otimes$ notation mostly used in $(\max, +)$ literature. For scalars x and y, $x \cdot y$ (with shorthand xy) denotes ordinary multiplication, not the $(\max, +)$ $\otimes$ operator. The max and $+$ operators are defined as usual with the additional convention that $-\infty$ is the zero-element of addition: $-\infty + x = x + -\infty = -\infty$ and the unit element of max, $\max(-\infty, x) = \max(x, -\infty) = x$. $(\max, +)$ is a *linear algebra*: $x + \max(y, z) = \max(x + y, x + z)$. The algebra is extended to a linear algebra of matrices and vectors as usual. Note that any matrix-vector multiplication in this paper denotes a $(\max, +)$ matrix-vector multiplication and not a traditional matrix-vector multiplication. For a matrix $\mathbf{M}$ and vector $\boldsymbol{x}$, we use $\mathbf{M}\boldsymbol{x}$ to denote this $(\max, +)$ matrix multiplication. If $\boldsymbol{a} = [a_i]$ and $\boldsymbol{b} = [b_i]$ with $a_i, b_i \in \mathbb{R}^{-\infty}$ are vectors of size k, then we write $\boldsymbol{a} \preceq \boldsymbol{b}$ to denote that for every $1 \leq i \leq k$, $a_i \leq b_i$. With $\boldsymbol{a}$ a vector and c a scalar, we use $c + \boldsymbol{a}$ or $\boldsymbol{a} + c$ to denote a vector with entries identical to the entries of $\boldsymbol{a}$ with c added to each of them: $c + \boldsymbol{a} = \boldsymbol{a} + c = [a_i + c]$. We use $\mathbf{0}$ to denote a vector with all zero-valued entries. The size of $\mathbf{0}$ is derived from the context. We use $\max(\boldsymbol{a}, \boldsymbol{b})$, defined as $[\max(a_i, b_i)]$ as a max operator on vectors and $\boldsymbol{a}+\boldsymbol{b}$, defined as $[a_i + b_i]$ as addition of vectors. $||\boldsymbol{a}||$ denotes a vector norm, defined as: $||\boldsymbol{a}|| = \max_i a_i$, i.e.,the maximum element. It is a proper vector norm in the algebra, because (i) $||\boldsymbol{a}|| = -\infty$ iff $a_i = -\infty$ for all i; (ii) $||c + \boldsymbol{a}|| = c + ||\boldsymbol{a}||$; (iii) $||\max(\boldsymbol{a}, \boldsymbol{b})|| \leq \max(||\boldsymbol{a}||, ||\boldsymbol{b}||)$. For a vector $\boldsymbol{a}$ with $||\boldsymbol{a}|| > -\infty$, we use $\boldsymbol{a}^{norm}$ to denote $\boldsymbol{a} - ||\boldsymbol{a}||$, the normalised vector $\boldsymbol{a}$, so that $||\boldsymbol{a}^{norm}|| = 0$. An inner product is defined as follows: $(\boldsymbol{a}, \boldsymbol{b}) := \max_i(a_i + b_i)$. If matrix $\mathbf{M} = [\boldsymbol{m}_j]$ (i.e., has column vectors $\boldsymbol{m}_j$), then $\mathbf{M}\boldsymbol{x} := \max_j(\boldsymbol{m}_j + \boldsymbol{x})$ and $\mathbf{M}^T\boldsymbol{x} := [(\boldsymbol{m}_j, \boldsymbol{x})]$. It is easy to verify that also matrix multiplication is linear: $\mathbf{M}(\max(\boldsymbol{x}, \boldsymbol{y})) = \max(\mathbf{M}\boldsymbol{x}, \mathbf{M}\boldsymbol{y})$ and $\mathbf{M}(c + \boldsymbol{x}) = c + \mathbf{M}\boldsymbol{x}$. Moreover, matrix multiplication is *monotone*: if $\boldsymbol{x} \preceq \boldsymbol{y}$, then $\mathbf{M}\boldsymbol{x} \preceq \mathbf{M}\boldsymbol{y}$.

A $(\max, +)$ automaton [6], recast in the context and terminology of this paper, is a 4-tuple $\mathcal{A} = (N, \boldsymbol{i}, \boldsymbol{f}, \mathcal{M})$. consisting of an integer number N of states, an initial vector $\boldsymbol{i}$, a final vector $\boldsymbol{f}$ and a morphism $\mathcal{M}$ on finite sequences of scenarios, mapping such sequences to an N by N matrix such that

$$\mathcal{M}(s_1 \ldots s_k) = \mathbf{G}_{s_k} \ldots \mathbf{G}_{s_1}.$$

The automaton associates with a sequence of scenarios, a completion time as follows:

$$\mathcal{A}(s_1 \ldots s_k) = (\boldsymbol{f}, \mathcal{M}(s_1 \ldots s_k)\boldsymbol{i}) = \boldsymbol{f}^T\mathbf{G}_{s_k} \ldots \mathbf{G}_{s_1}\boldsymbol{i}.$$

In this way, $\boldsymbol{i}$ captures the initial enabling times of the graph's initial tokens. Usually it is assumed that $\boldsymbol{i} = \mathbf{0}$. Then, $\mathcal{M}(\overline{s})\boldsymbol{i}$ captures the production times of the initial tokens of the SADF after the sequence $\overline{s}$ of scenarios. The final vector $\boldsymbol{f}$ specifies how this final vector determines the metric we are interested in. If this is for instance the usual concept of makespan, then this can be captured by taking $\boldsymbol{f} = \mathbf{0}$. Oftentimes we are interested in worst-case increase of $\mathcal{A}(\overline{s})$ for growing length of $\overline{s}$. In our case, this represents the worst-case throughput for any sequence of scenarios. [6] shows how this maximum growth rate, determining minimum throughput can be efficiently computed as the maximum cycle mean (MCM) of the equivalent timed event graph [1] of the matrix $\mathbf{G} = \max_{s \in S} \mathbf{G}_s$. It also shows how, given an infinite regular sub language of S^*, the set of all finite scenario sequences, the maximum growth rate can be determined by a product of automata. We exploit this result to determine throughput of an SADF for a given FSM specifying the scenario sequences.

3.3 Scenario-Aware Dataflow

Every individual scenario is modelled by an SDFG in the form of its corresponding $(\max, +)$ matrix. See [9] for a method to derive the $(\max, +)$ matrix using a symbolic execution of a single iteration of the SDF graph. An SADF graph is further characterised by the possible orders in which certain scenarios may occur. SADF specifies scenario sequences by means of a Finite State Machine (FSM). (In [25] scenario sequences are stochastically characterised by a Markov chain. For worst-case analysis we can abstract from transition probabilities and obtain an FSM.) Every state is labelled with a scenario and different states can be labelled with the same scenario (See Figure 4). Since we focus here on streaming applications, we consider *infinite* executions of the FSM to characterise the scenario sequences that may occur. Note that since our worst-case performance requirements are safety requirements, there is no need for specific acceptance conditions, such as, e.g., Büchi automata.

Definition 1. *Given a set S of scenarios. A scenario finite state machine F on S is a tuple (Q, q_0, δ, Σ) consisting of a finite set Q of states, an initial state $q_0 \in Q$, a transition* *relation $\delta \subseteq Q \times Q$ and a scenario labelling $\Sigma : Q \to S$. A path of F is a sequence $\overline{q}$ of states $\overline{q}(k) \in Q$ with $\overline{q}(0) = q_0$ such that for all $k \geq 0$, $(\overline{q}(k), \overline{q}(k+1)) \in \delta$.*

Every path $\overline{q}$ through the FSM corresponds to a sequence $\overline{s}$ of scenarios with $\overline{s}(k) = \Sigma(\overline{q}(k))$.

4. A STATE-SPACE EXECUTION MODEL

The FSM specifies all possible orders in which scenario sequences can occur. With that sequence of scenarios and an initial time-stamp vector $\boldsymbol{\gamma}_0$, we can associate a sequence $\boldsymbol{\gamma}_0\boldsymbol{\gamma}_1\boldsymbol{\gamma}_2\ldots$ of vectors with $\boldsymbol{\gamma}_{k+1} = \mathbf{G}_{s_{k+1}}\boldsymbol{\gamma}_k$. We derive straightforwardly from the theory of SDF [23] that this sequence of vectors is guaranteed to be an upper bound on the self-timed execution of the dataflow system's execution under this scenario sequence. Iteration k is guaranteed completed at the latest according to time-stamp vector $\boldsymbol{\gamma}_k$, but possibly earlier.

Definition 2. (Throughput) *Throughput of an SADF is defined as the largest value $\tau \in \mathbb{R}$ such that for every possible scenario sequence and corresponding vector sequence $\overline{\boldsymbol{\gamma}}$, for every $\epsilon \in \mathbb{R}$, $\epsilon > 0$, there is some $K \in \mathbb{N}$ such that for all $L \in \mathbb{N}$, $L > K$, $\frac{L}{||\overline{\boldsymbol{\gamma}}(L)||} > \tau - \epsilon$.*

Note that the definition roughly states that the throughput achieved by the specific scenario sequence is always at least τ. The somewhat cumbersome definition is used since not for every scenario sequence, the limit of the average throughput, which might seem a more intuitive definition, exists. Here we have defined throughput in terms of the number of iterations, which is in accordance with the definition of throughput of SDF as used in [10]. However, it is straightforward to adapt the definition of throughput to for instance the average number of firings of a particular actor or the average number of tokens produced on a particular channel, even if that number differs per scenario, as explained below.

Similarly, one can define latency of the SADF graph.

Definition 3. (Latency) *The latency of an SADF relative to desired period $\pi \in \mathbb{R}$ is defined as the smallest vector $\boldsymbol{\lambda}$ such that for every possible scenario sequence and corresponding vector sequence $\overline{\boldsymbol{\gamma}}$, for every $k \leq 0$, $\overline{\boldsymbol{\gamma}}(k) \preceq k\pi + \boldsymbol{\lambda}$.*

To compute throughput and latency, we have to check all possible scenario sequences. To do this, we first define a state-space of scenario sequence executions. From the FSM $F = (Q, q_0, \delta, \Sigma)$ and the set of SDF matrices $\{\mathbf{G}_s \mid s \in S\}$ we can define a new state space (C, c_0, Δ) as follows.

- A set $C = Q \times \mathbb{R}^{-\infty N}$ of configurations $(q, \boldsymbol{\gamma})$ with a state $q \in Q$ of F and a $(\max, +)$ vector $\boldsymbol{\gamma}$.
- An initial configuration $c_0 = (q_0, \mathbf{0})$.
- A labelled transition relation $\Delta \subseteq C \times \mathbb{R} \times C$ consisting of the following transitions: $\{((q, \boldsymbol{\gamma}), ||\boldsymbol{\gamma}'||, (q', \boldsymbol{\gamma}'^{norm})) \mid (q, \boldsymbol{\gamma}) \in C, (q, q') \in \delta, \boldsymbol{\gamma}' = \mathbf{G}_{\Sigma(q')}\boldsymbol{\gamma}\}$.

A state of the new state space is a pair consisting of a state of the scenario FSM and a normalised vector indicating the relative distance in time of the time-stamps of the tokens. An edge in this state space represents the execution of a single iteration in the scenario indicated by the FSM state of the destination of the edge. The meaning of an edge $((q_1, \boldsymbol{\gamma}_1), d, (q_2, \boldsymbol{\gamma}_2))$ can be clarified as follows. If we start with tokens having time-stamps according to $\boldsymbol{\gamma}_1$ and we execute a single iteration in scenario $\Sigma(q_2)$ then the new set of tokens are guaranteed to be ultimately produced at $d + \boldsymbol{\gamma}_2$ (or earlier). Note that because of linearity, this means that if the starting vector equals $t + \boldsymbol{\gamma}_1$, then an upper bound for the new vector is $t + d + \boldsymbol{\gamma}_2$. In general, for any path leading to state $(q, \boldsymbol{\gamma})$, the exact tightest upper bound that can be given is $T + \boldsymbol{\gamma}$ where T equals the sum of the delays along the edges of the path.

The possible behaviours of our SADF graph are captured by the reachable part of this state space. It can be constructed in depth-first-search (DFS) or breadth-first-search (BFS) (or any other) manner incrementally while checking for desired constraints such as latency.

The reachable state space of the example graph is shown in Figure 7. The bold state is the initial state. The grey box highlights the cycle with lowest throughput. It is achieved only by an alternation of scenarios a and b. Bold arrows indicate a possible path (scenario sequence) to this cycle. The overall throughput can be determined by a maximum cycle mean analysis of the state-space, but we discuss a more efficient method later on. Assuming we know that the maximum achievable throughput is $1/3$ (the minimum period is 3), we can determine the latency in a single traversal of the state space. We demonstrate this only along the path indicated with bold arrows. We have to determine the smallest $\boldsymbol{\lambda}$ such that $\boldsymbol{\gamma}_k \preceq \boldsymbol{\lambda} + 3k$. In other words,

$$\boldsymbol{\lambda} = \max_k\{\boldsymbol{\gamma}_k - 3k\}$$

$$\boldsymbol{\lambda} = \max_k \left\{ \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 3 \\ 3 \\ 2 \end{bmatrix} - 3, \begin{bmatrix} 4 \\ 4 \\ 6 \end{bmatrix} - 6, \begin{bmatrix} 9 \\ 9 \\ 6 \end{bmatrix} - 9 \right\}$$

So, for this example (also if we complete it for the entire state space), the latency is simply given by the vector $\boldsymbol{\lambda} = [0;\ 0;\ 0]^T$. If we are specifically interested in the completion time of actor C, its latency can be directly derived from $\boldsymbol{\lambda}$ as follows. In scenario a, by the vector inner product $([-\infty;\ -\infty;\ 2]^T, \boldsymbol{\lambda})$ and in scenario b by $([-\infty;\ -\infty;\ 1]^T, \boldsymbol{\lambda})$. With the above solution of λ this guarantees a latency of 2 for any firing of actor C.

We can give a practical condition under which the state space is finite.

Proposition 4.1. *If for every possible scenario sequence $\overline{s}$ allowed by the FSM and any $k \geq 0$ there is some $m > k$ such that the matrix $\mathbf{H} = \prod_{k \leq l < m} \mathbf{G}_{\overline{s}(l)}$ contains no entries $-\infty$, then the reachable part of the state space is finite.*

Proof. We only give the intuition behind the condition here. An entry $-\infty$ means there is no dependency. If in a particular scenario sequence it is possible for parts of the graph to evolve completely independently from other parts, then the graph may become unbounded, because the faster time-stamps will diverge to $-\infty$. Otherwise any token can be shown to be dependent on a critical token by a path of bounded length and the time-stamps can only take a finite number of different values. □

Proposition 4.2. *The throughput of an SADF is equal to the inverse of the maximum cycle mean attained in any of the reachable simple cycles of the state space.*

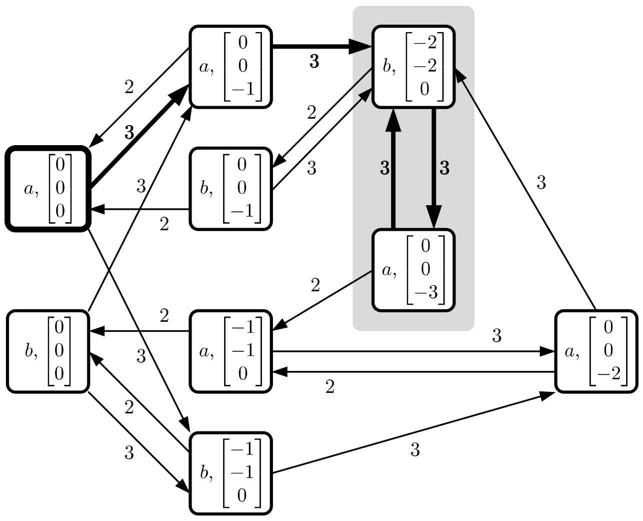


Figure 7: State space of example SADF graph

Proof. (Sketch.) Let τ be the throughput of the SADF and let μ be the maximum cycle mean among the reachable simple cycles. A scenario sequence that goes to the cycle of μ and then cycles around it has a throughput of $1/\mu$. Therefore, $\tau \leq 1/\mu$. Assume that $\tau > 1/\mu$, then there exists a sequence of scenarios such that the average throughput remains in an ϵ-environment of τ that does not include $1/\mu$. Because the reachable state space is finite, there must be a cycle with a cycle mean larger than μ and hence also a simple cycle with a cycle mean larger than μ. That would contradict the definition of μ and hence, $\tau = 1/\mu$. $\square$

The grey box in Figure 7 highlights the cycle with the maximum cycle mean which determines the throughput and the bold arrows indicate a possible path (scenario sequence) to the critical cycle. We mentioned earlier that it is possible to measure throughput in terms of the average number of actor firings or produced tokens per time unit if that number varies per scenario. In that case, one would annotate the transitions additionally with the number of firings performed or tokens produced in that particular iteration and compute the Maximum Cycle Ratio (MCR) of both annotations, which is equally efficient using Karp's method [15] The calculation of the latency vector is similarly adjusted for the number of firings instead of the number of iterations only.

5. WORST-CASE THROUGHPUT

The state space of the SADF graph can be large and we discuss a reduction technique in the following section, but first we focus on worst-case throughput calculation. For this, it is not necessary to construct the state space explicitly. Instead we may work directly on the $(\max, +)$ automaton which generates the explicit state space, using the results of Gaubert [6]. N is the number of initial tokens of the SADF with scenarios S, the order of which is specified by the FSM (Q, q_0, δ, Σ). Let $\mathbf{G}_s$ be the $(\max, +)$ matrix of scenario s.

Then we can define directly the *throughput graph* (V, E), which can be obtained in a number of steps from the FSM and the scenario matrices using procedures from [6]. It has a set V of vertices each of which represents a combination of

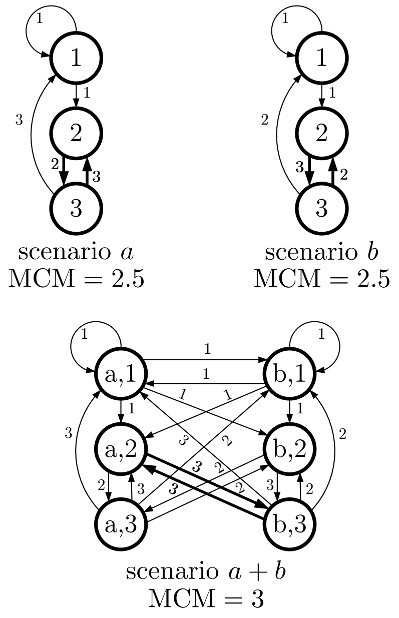


Figure 8: $(\max, +)$ automata of example SADF graph

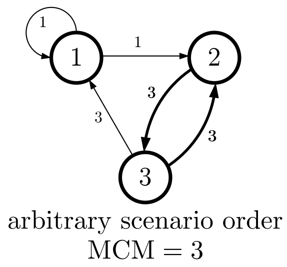


Figure 9: Reduced $(\max, +)$ automaton of example SADF graph

one of the initial tokens and a location of the FSM with its implied scenario. The set E of labelled edges represents the worst-case time dependencies between the individual tokens in the specific locations and corresponding scenarios. It is formally defined as follows.

- $V = \{(q, n) \mid q \in Q, 1 \leq n \leq N\}$
- $E = \{((q_1, m), d, (q_2, n)) \mid q_1, q_2 \in Q, \mathbf{G}_{\Sigma(q_2)}(m, n) = d \neq -\infty, 1 \leq m, n \leq N\}$

The resulting graph for the running example is shown as the bottom graph in Figure 8. It can be seen as being constructed from the individual timed event graph representations of the individual scenarios [1]. The upper two graphs show them for the scenarios a and b. Observe that they are equivalent representations of the scenario matrices $\mathbf{G}_a$ and $\mathbf{G}_b$ respectively; there is an edge from node m to node n if the corresponding entry at row m and column n of the matrix is unequal to $-\infty$ and it carries the same value. (The MCMs of these graphs correspond to the largest eigenvalues of the matrices and determine the maximal throughput within the particular scenario.) The final throughput graph can be constructed from these graphs and the FSM of the SADF by replacing every location of the FSM by the graph of its corresponding scenario and for every edge in the FSM, connect the time dependency edges to the corresponding initial tokens of the corresponding successor scenario graph. Applying this to the FSM in Figure 1 and the two graphs at the top of Figure 8 gives the result shown at the bottom of Figure 8.

Observe that the size of the throughput graph is as follows. It has $|Q| \times N$ vertices, i.e., the number of states in the FSM times the number of initial tokens in the graph. And at most $|V|^2$ edges. This graph is typically much smaller than the explicit state space. Based on the throughput graph, the worst-case throughput of the SADF can be determined as the MCM of the graph; the MCM analysis is $O(|V| \cdot |E|) = O((|Q| \cdot N)^3)$. Correctness of the method follows from the results of [6].

Proposition 5.1. *The throughput of an SADF is equal to the inverse of the maximum cycle mean of its throughput graph.*

Proof. Follows straightforwardly by construction of the graph, the $(\max, +)$ automaton semantics of SADF and Proposition 1 in [6]. □

In many cases in practice, the scenarios may occur in arbitrary order in which case the scenario FSM is fully connected with a single state for every scenario and the throughput graph is very regular, simplifying the analysis as follows.

Proposition 5.2. *The throughput of an SADF with a fully connected FSM is equal to the inverse of the maximum cycle mean of the timed event graph corresponding to the matrix* $\mathbf{G} = \max_{q \in Q} \mathbf{G}_{\Sigma(q)}$.

Proof. Follows straightforwardly from the $(\max, +)$ automaton semantics of SADF and Theorem 2 of [6]. □

In this case one can construct a simplified throughput graph with one vertex for every initial token only. For the example graph we have (using $\mathbf{G}_a$ and $\mathbf{G}_b$ as in Section 3.2).

$$\mathbf{G} = \max(\mathbf{G}_a, \mathbf{G}_b) = \begin{bmatrix} 1 & -\infty & 3 \\ 1 & -\infty & 3 \\ -\infty & 3 & -\infty \end{bmatrix}$$

The timed event graph corresponding to $\mathbf{G}$ is shown in Figure 9 and has an MCM of 3, the eigenvalue of $\mathbf{G}$.

It is important to repeat that although we are allowed in this situation to construct a new matrix taking the maximum of the elements of the individual scenario matrices, this does not mean that we could have achieved this result by either considering only the 'worst-case' scenario or by considering the graph in which each actor takes as its execution time the worst-case over all scenarios. One can easily demonstrate that the former approach is strictly too optimistic (as the running example illustrates) and the latter approach is strictly too pessimistic.

6. STATE SPACE ANALYSIS

We have implemented an algorithm that computes the execution state space of the SADF graph. We use a basic breadth-first-search approach for this. It is given as Algorithm 1. It constructs the state space StSp straightforwardly according to the definition of Section 4. The while loop realises a breadth-first exploration by keeping unexplored states in a queue BFS.

Algorithm 1 Compute state space of an SADF

```
ComputeStateSpace(G)
Input: SADF  G
BFS := new Queue()
StSp := new Graph()
BFS.insert((G.FSM.$q_0$, $\mathbf{0}$))
StSp.insertState((G.FSM.$q_0$, $\mathbf{0}$))
while not BFS.isEmpty() do
    $(q, \boldsymbol{\gamma})$ := BFS.removeBegin()
    for $q'$ in $q$.nextStates() do
        $\boldsymbol{\gamma}'$ := G.execute($\boldsymbol{\gamma}$, $q'$.scenario);
        StSp.addEdge(($q$, $\boldsymbol{\gamma}$), $\boldsymbol{\gamma}'$.norm(), ($q'$, $\boldsymbol{\gamma}'$.normalize()))
        if not StSp.includesState($q'$, $\boldsymbol{\gamma}'$) then
            StSp.insertState($q$, $\boldsymbol{\gamma}'$)
            BFS.add($q'$, $\boldsymbol{\gamma}'$.normalize())
        end if
    end for
end while
return StSp
```

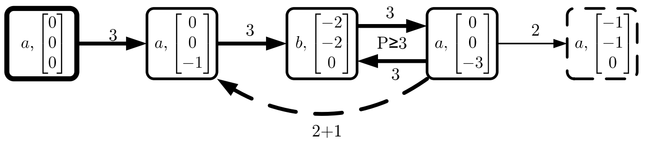


Figure 10: Reduction by approximating the behaviour

During the exploration procedure, we enumerate the state-space and check timing constraints or compute latency. Exploration continues until we reach a state that we visited before, we back-track and stop exploration in that direction. Still, the state space can be large and using the information in the time-stamp vectors, we can do a bit more to limit the space we need to search. We can try to create a an edge to a different state, already visited before with the same FSM state, but different vector, with a conservative (pessimistic) estimate of the timing guarantees. This might be sufficient for instance if we run the algorithm to show that the graph can attain a certain latency, or if we want to trade-off the run-time for an approximate answer.

The approach is illustrated in Figure 10. It shows a state-space exploration in progress of the example state space of Figure 7. The latest state explored is $(a, [-1; -1; 0]^T)$ at the right-hand side of the figure. This is not yet a recurrent state and we can decide to continue exploration. We also have the option however of closing the cycle to state $(a, [0; 0; -1]^T)$ already in the set of visited states, indicated by the dashed arrow. Knowing that in general $\boldsymbol{\gamma}_2 \preceq \boldsymbol{\gamma}_1 + ||\boldsymbol{\gamma}_2 - \boldsymbol{\gamma}_1||$, we are guaranteed that an execution according to the given cycle can be done with a period of at most the sum of delays in the cycle, incremented with $||\boldsymbol{\gamma}_2 - \boldsymbol{\gamma}_1||$. In this case, the norm of the difference is 1 and the sum of the delays equals 5 for 3 iterations, so the throughput for this cycle is conservatively estimated to be at least 3/6 while in reality it is 6/9.

If we are not trying to compute for instance the minimal latency vector, but we are trying to prove that a certain given latency can be attained, we can try to close cycles using the rule above as soon as possible and only if it does not enable us to prove that the required latency can be attained, then we continue.

Algorithm 1 has been implemented in our publicly available SDF^3 software library for SDF analysis [24]. We applied an some additional optimization to the algorithm which is optimized for presentation. We have used edge-labelled FSMs instead of the state-labelled, since they tend to have fewer states. Conversion between one and the other approach is trivial.

7. EXPERIMENTAL RESULTS

We have done experiments with the MP3 model discussed earlier, on an H.263 model and on a large collection of randomly generated graphs, on a 3.4GHz, Intel Pentium 4 based PC. A conservative SDF model, without scenarios, of the MP3 decoder can guarantee a throughput of $1.08 \cdot 10^{-7}$ frames per cycle. Experiments with the SADF graph show that the MP3 graph with frame type scenarios can achieve a guaranteed throughput of $1.72 \cdot 10^{-7}$ frames per processors cycle, 59% higher. Since one frame equals 26ms of real time, the application can make its real-time requirements if the processor performs at least 1/0.026 frames per second divided by $1.72 \cdot 10^{-7}$ frames per cycle, equals 224 MCycles per second. The corresponding state space of the MP3 SADF has only 17 states and the analysis took less than 10ms. In comparison, a similar MP3 model from the web site of [24] took 21.49s to analyse for average throughput with the tool of [25].

We have also used our method on the H.263 decoder graph used in [25] (Figure 3). The run-time of our algorithm on this graph with 5 actors is around 10ms. It has 121 states. Increasing the pipeline depth (adding initial tokens to the edge from RC to FD) to 5, increases the time for generating the state space (now 1093 states) to 160ms, but the throughput can still be analysed with the method of Section 5 in 10ms.

We have also analysed a collection of more than 5000 random SADF graphs between 3 and 18 actors with an entry in the repetition vector between 1 and 7 and between 10 and 70 scenarios, giving an average 36% higher throughput guarantee than from a conservative SDF model of the same graph. The execution times for state space generation are summarised in Figure 11, averaged for graphs with a particular number of actors (a), a particular number of scenarios (b) or a particular maximum entry in the repetition vector (c). The graph shows that the average execution times scale with the complexity of the graph. They scale roughly linear with the number of actors and with the number of scenarios. The execution time is most strongly correlated with the number of scenarios. It scales more than linear with the repetition vector entries. These results are mostly due to the conversion from the SDF scenario graphs to $(\max, +)$ matrices, during which an iteration of the SDF graph is symbolically simulated.

8. FUTURE WORK AND CONCLUSIONS

An interesting future application of the approach is possible when in the FSM characterising the possible scenario sequences we can distinguish *application scenarios* from *system scenarios*. Application scenarios are scenarios of behaviour that are forced upon the system from the application behaviour. For instance, in the different MP3 frame types, determined by the MP3 file, not by the system. System scenarios, in contrast, may be behaviours selected by the system itself, depending on conditions and optimisation criteria. For instance resource allocation or voltage/frquency scaling. This may give rise to a game-theoretic approach to scenario determination by an FSM, where a good system controller can be defined in terms of a winning strategy in a game where every move of the environment (application scenario) needs to be countered with a system move (selection of a system scenario) so as to achieve an overall goal (for instance minimising power consumption while guaranteeing throughput and latency constraints). It is a well studied subject in automata theory and techniques exist to automatically generate such controllers [27]. The potential optimisation technique discussed in Section 6 needs to be investigated, which requires an efficient way of finding good states to close the cycles.

To conclude, in this paper we have introduced a formal model of Scenario-Aware Dataflow graphs based on the theory of $(\max, +)$ automata and derive from this theory techniques to analyse worst-case performance. The techniques can give the exact highest throughput that can be guaranteed for the specific model very efficiently and can determine minimal latency that can be attained from a state space analysis. An algorithm and implementation of the techniques have been presented and have been experimentally evaluated, showing that it can give much tighter performance guarantees for many SADF graphs than SDF based analysis and that run-times are often good, in particular for throughput analysis without state-space generation.

Acknowledgment

This work is supported by the EC through FP7 IST project 216224, MNEMEE.

9. REFERENCES

[1] F. Baccelli, G. Cohen, G. Olsder, and J.P.Quadrat. *Synchronization and Linearity.* John Wiley & Sons, 1992.

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

[3] M. Brilman and J.-M. Vincent. Dynamics of synchronized parallel systems. *Stochastic Models*, 13(3):605–617, 1997.

[4] J. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model.* PhD thesis, University of California, EECS Dept., Berkeley, CA, 1993.

[5] J. Q. G. Cohen, D. Dubois and M. Viot. A linear system-theoretic view of discrete-event processes and its use for performance evaluation in manufacturing. *IEEE Trans. Automatic Control*, 30:210–220, 1985.

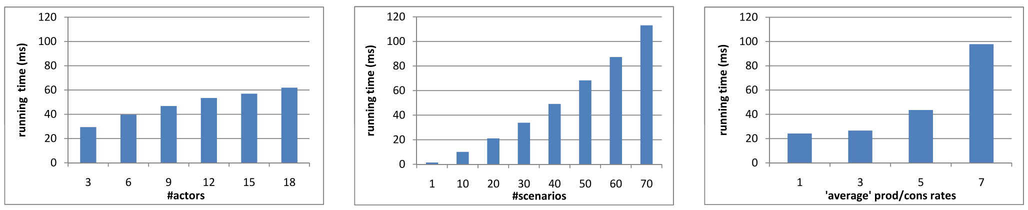
[6] S. Gaubert. Performance evaluation of $(\max, +)$ automata. *IEEE Trans. Automatic Control*, 40(12):2014–2025, 1995.

[7] S. Gaubert and J. Mairesse. Modeling and analysis of timed petri nets using heaps of pieces. *IEEE Trans. Automatic Control*, 44(4):683–697, April 1999.

[8] M. Geilen. Reduction techniques for synchronous dataflow graphs. In *Proc. of 46th Design Automation Conference, DAC 2009*, 2009.

[9] M. Geilen. Synchronous data flow scenarios. *Transactions on Embedded Computing Systems, Special issue on Model-driven Embedded-system Design, to be published*, 2009.

[10] A. Ghamarian, M. Geilen, S. Stuijk, T. Basten, A. Moonen, M. Bekooij, B. Theelen, and M. Mousavi. Throughput analysis of synchronous data flow graphs. In *Proceedings of the 6th International Conference on Application of*



(a) #actors (b) #scenarios (c) #rep.vec.

Figure 11: Average execution times in ms for random graphs

Concurrency to System Design (ACSD'06), pages 27–30. IEEE Computer Society Press, Los Alamitos, CA, USA, 2006.

[11] A. Girault, B. Lee, and E. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, 18(6):742–760, June 1999.

[12] R. Henia and R. Ernst. Scenario aware analysis for complex event models and distributed systems. In *RTSS '07: Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pages 171–180, Washington, DC, USA, 2007. IEEE Computer Society.

[13] G. Kahn. The semantics of a simple language for parallel programming. In J. Rosenfeld, editor, *Information Processing 74: Proceedings of the IFIP Congress 74, Stockholm, Sweden, August 1974*, pages 471–475. North-Holland, Amsterdam, Netherlands, 1974.

[14] G. Kahn and D. MacQueen. Coroutines and networks of parallel programming. In B. Gilchrist, editor, *Information Processing 77: Proceedings of the IFIP Congress 77, Toronto, Canada, August 8-12, 1977*, pages 993–998. North-Holland, 1977.

[15] R. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics*, 23(3):309–311, 1978.

[16] J. Keinert, H. Dutta, F. Hannig, C. Haubelt, and J. Teich. Model-based synthesis and optimization of static multi-rate image processing algorithms. In *Proceedings of Design, Automation & Test in Europe*, pages 135–140, 2009.

[17] E. Lee and D. Messerschmitt. Synchronous data flow. *IEEE Proceedings*, 75(9):1235–1245, Sept. 1987.

[18] S. Mamagkakis, D. Soudris, and F. Catthoor. Middleware design optimization of wireless protocols based on the exploitation of dynamic input patterns. In *DATE '07: Proc. of the conference on Design, automation and test in Europe*, pages 1036–1041, San Jose, CA, USA, 2007. EDA Consortium.

[19] L. T. X. Phan, S. Chakraborty, and P. S. Thiagarajan. A multi-mode real-time calculus. In *RTSS '08: Proceedings of the 2008 Real-Time Systems Symposium*, pages 59–69, Washington, DC, USA, 2008. IEEE Computer Society.

[20] P. Poplavko, T. Basten, and J. van Meerbergen. Execution-time prediction for dynamic streaming applications with task-level parallelism. In *DSD '07: Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools*, pages 228–235, Washington, DC, USA, 2007. IEEE Computer Society.

[21] P. Poplavko, M. Geilen, and T. Basten. Predicting the throughput of multiprocessor applications under dynamic workload. In *Proceedings of ICCD-2010, the International Conference on Computer Design*, 2010.

[22] N. Scaife and P. Caspi. Integrating model-based design and preemptive scheduling in mixed time- and event-triggered systems. In *ECRTS '04: Proceedings of the 16th Euromicro Conference on Real-Time Systems*, pages 119–126, Washington, DC, USA, 2004. IEEE Computer Society.

[23] S. Sriram and S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., New York, NY, USA, 2000.

[24] S. Stuijk, M. Geilen, and T. Basten. SDF^3: SDF For Free. In *Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings*, pages 276–278. IEEE Computer Society Press, Los Alamitos, CA, USA, June 2006.

[25] B. D. Theelen, M. Geilen, T. Basten, J. Voeten, S. V. Gheorghita, and S. Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *MEMOCODE*, pages 185–194, 2006.

[26] L. Thiele and N. Stoimenov. Modular performance analysis of cyclic dataflow graphs. In *EMSOFT '09: Proceedings of the seventh ACM international conference on Embedded software*, pages 127–136, New York, NY, USA, 2009. ACM.

[27] W. Thomas. On the synthesis of strategies in infinite games. In *STACS*, pages 1–13, 1995.

[28] G. Viennot. *Combinatoire Énumérative*, chapter Heaps of Pieces, I: Basic definitions and combinatorial lemmas, pages 321–350. Springer, 1986.

[29] P. Yang, P. Marchal, C. Wong, S. Himpe, F. Catthoor, P. David, J. Vounckx, and R. Lauwereins. Managing dynamic concurrent tasks in embedded real-time multimedia systems. In *ISSS '02: Proceedings of the 15th international symposium on System Synthesis*, pages 112–119, New York, NY, USA, 2002. ACM.