Synchronous dataflow scenarios

Synchronous Data Flow Scenarios[1]

Marc Geilen
Eindhoven University of Technology
Eindhoven, The Netherlands
m.c.w.geilen@tue.nl

The Synchronous Data-Flow (SDF) model of computation by Lee and Messerschmitt has become popular for modelling concurrent applications on a multi-processor platform. It is used to obtain a guaranteed, predictable performance. The model is on the other hand quite restrictive in its expressivity, making it less applicable to many modern, more dynamic applications. A common technique to deal with dynamic behaviour is to consider different scenarios in separation. This analysis is however, currently limited mainly to sequential applications.

In this paper, we present a new analysis approach that allows analysis of Synchronous Data-Flow models across different scenarios of operation. The dataflow graphs corresponding to the different scenarios can be completely different. Execution times, consumption and production rates and the structure of the SDF may change. Our technique allows to derive or prove worst-case performance guarantees of the resulting model and as such extends the model-driven approach to designing predictable systems to significantly more dynamic applications and platforms. The approach is illustrated with three MP3 and MPEG-4 related case studies.

Categories and Subject Descriptors: F.1.1 [**Computation by Abstract Devices**]: Models of Computation

1. INTRODUCTION

Design of embedded systems is becoming increasingly complex because of larger numbers of use-cases, heterogeneous multiprocessor platforms and increasingly complex and dynamic applications. In order to manage this complexity, a number of trends emerge. Model-based design [Lee and Sangiovanni-Vincentelli 1998; Thiele et al. 2000] strives to design platforms and applications in such a way that systems can be designed based on simple, abstract models and such that the (automatically) realised design is guaranteed to behave according to the model with a predictable performance. Platforms should be as *composable* as possible [Sangiovanni-Vincentelli and Martin 2001] and the application model simple enough to allow it to be analysed and to serve as the starting point for a synthesis trajectory. Relatively simple and static models are used for this, such as (homogeneous) synchronous data-flow graphs or finite state automata. Modern applications however are more dynamic than can be captured by such models without incurring too much overestimation and too pessimistic assumptions. Models used in this process often capture aspects of both the application behaviour as well as of the platform on which the applications are mapped and of the impact of other applications contending for the resources and the arbitration of those resources.

[1]This work was partially supported by the Netherlands Ministry of Economic Affairs under the Bisk program in the Octopus project and by the European Commission under the FP7 program in the ICT-216224 project MNEMEE.

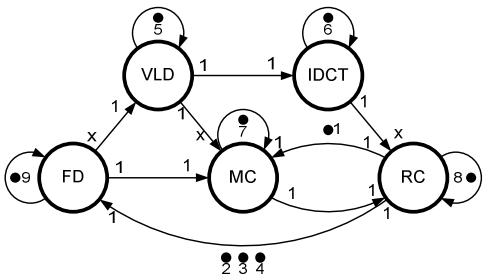

Fig. 1. Scenario graphs of MPEG-4 SP decoder

A second emerging trend is *scenario-based design* [Yang et al. 2002; Gheorghita et al. 2006; Mamagkakis et al. 2007]. In this approach, the dynamic behaviour of an application is viewed upon as a collection of different behaviours (*scenarios*) occurring in some arbitrary order, 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 scenarios and dealing with scenario transitions.

In this paper we introduce techniques to model and analyse systems, the behaviour of which can be captured as a collection of scenarios, each expressed by means of a Synchronous Data-Flow graph. Every aspect of the data-flow graphs may be different in different scenarios, such as execution times, communication rates or graph structure. Dependencies between subsequent scenarios are captured by tokens which are transferred from one graph to the next. We show that the model naturally captures pipelining of subsequent scenarios. As a mathematical basis for our techniques, we use Max-Plus algebra. It provides the mathematical tools for systems based on synchronisation and delays.

Individual scenarios are synchronous data-flow graphs. Analysis methods for SDFGs exist and have recently been extended [Sriram and Bhattacharyya 2000; Ghamarian et al. 2008; Wiggers et al. 2007]. In this paper we introduce an algorithm to analyse the transient and steady-state behaviour of an SDFG and thus enable the analysis of the changes from one scenario into the next.

As an example, we have a closer look at an MPEG-4 Simple Profile decoder [Theelen et al. 2006] in Figure 1 (details of the graphical notation that are not important here, are explained in Section 3). It consists of the following components. A stream parser FD (frame detector). VLD is a variable-length decoder decompressing the entropy-coded stream of macro-blocks. MC is motion compensation, RC reconstruction of the output frame and IDCT the inverse discrete cosine transformation. Frames can be classified in different types. Primarily, there is a difference between I-frames and P-frames. I-frames are encoded independently from previous frames, while P-frames exploit temporal correlation between subsequent frames, but they need a reference to a previous frame. Decoding of I and P-frames is partly different. Among P frames however, there is still a large variation in workload. For a QCIF size image, there can be anywhere between 0 and 99 macro blocks per frame. Together with the frame type, this number can be used as a classification for scenarios. For this example, the following scenarios are identified. The first one

is decoding of an I-frame, it does not contain any motion vectors. For such a frame, 99 macro blocks are processed by VLD and IDCT and go into the reconstructed frame, while MC is not used. The other scenarios correspond to P-frames. In a P frame, the new frame is constructed from blocks from the previous frame, corrected by a motion vector, together with a number of new blocks. The various scenarios represent different numbers of macro blocks. Scenario 'P30' represents the case of 0 to 30 new blocks, modelled with the worst-case situation within that scenario, namely 30 blocks. Similarly, scenario P40 represents 31-40 blocks and so on for P50, P60, P70, P80 and P99. The VLD and IDCT operations are performed for every individual block and the other operations once per frame. Hence, also the communication rates vary with the scenario (x in Figure 1, representing the Px scenarios, for the I scenario the edge from VLD to MC does not exist). The execution times of the operations are also represented by the worst-case situation per scenario. We would like to answer the following kind of questions. Can the graph meet throughput and latency constraints irrespective of the order of frame scenarios? Given a sequence of frame scenarios, can we make an accurate predication of the decoding completion time? Can we do dynamic voltage and frequency scaling based on scenario information and still guarantee to meet deadlines for this and following scenarios? With traditional methods for analysing SDFGs, these questions cannot be answered, or they can only be answered with a high conservative margin and overestimation. In this paper we show how such questions can be answered more satisfactorily.

The rest of the paper consists of the following parts. In the next section we discuss related work. Then we give some mathematical preliminaries of SDF and Max-Plus algebra. Section 5 introduces an algorithm to analyse the transient behaviour of an SDFG. Section 6 explores different types of analysis enabled by the new algorithm. The methods are illustrated with case studies in Section 7, followed by concluding discussion.

2. RELATED WORK

Current state-of-the-art in model-based design of embedded multiprocessor system software and platforms exploits a spectrum of models, ranging from statically analysable models for guaranteed performance including models of limited expressivity such as SDF [Lee and Messerschmitt 1987] and finite automata to more expressive models such as Kahn Process Networks [Kahn 1974; Buck 1993; Stefanov et al. 2004] and combinations of models [Künzli et al. 2007; Lee and Sangiovanni-Vincentelli 1998], which are more expressive and flexible, but often not statically analysable for hard guarantees. This paper extends that spectrum with a model that is more dynamic and flexible than the former and thus allows for tighter estimations and still provides hard guarantees in contrast with the latter.

We started from the tradition of using Synchronous Data-Flow models [Sriram and Bhattacharyya 2000; Ghamarian et al. 2008; Wiggers et al. 2007]. Some extensions of the model are in use, such as Cyclo-Static Data-Flow (CSDF, [Bilsen et al. 1996; Wiggers et al. 2007]), which can capture changes in rates or execution times in a data-flow graph, but only changes in fixed repetitive patterns. A per-scenario analysis allows a tighter estimate of worst-case behaviour. More expressive data-

flow models such as Boolean Data-Flow or Dynamic Data-Flow cannot be statically analysed [Buck 1993]. Hybrid automata-SDF models have been studied, such as HDF [Girault et al. 1999] but only in the context of sequential schedules. It therefore does not consider pipelined execution and the implications of pipelined scenario transitions as we do in this paper. The results obtained in this paper directly apply to the HDF model of computation. [Bhattacharya and Bhattacharyya 2001] introduces a parameterised data-flow model, where parameter values can be interpreted as different scenarios. The focus of that work however is on sequential software synthesis rather than multiprocessor implementation.

The additional expressivity and flexibility is derived from considering scenarios separately. State-of-the-art scenario-based analysis [Gheorghita et al. 2006; Yang et al. 2002; Mamagkakis et al. 2007] mostly deals with sequential code and not with parallel applications such as SDF graphs. In [Mamagkakis et al. 2007], scenarios are applied from a system perspective on a concrete design case. Scenario based-analysis has been applied to homogeneous data-flow graphs in [Poplavko et al. 2007] with varying execution times per scenario. [Poplavko et al. 2007] proposes that the number and types of scenario switches over a period to follow are embedded in a video stream by the encoding party. It is shown that for an MPEG-4 Advanced Video Coding scheme, this yields a much improved estimation of the worst-case execution time at affordable overhead cost. In this paper, we generalise this analysis in several ways. Our technique is also directly applicable to non-homogeneous SDF graphs and we additionally allow production and consumption rates to vary between scenarios and allow the graph structure to change. Moreover, we have a more flexible characterisation of the transient behaviour between scenarios by considering different combinations of a delay and period to be used as a bound on the behaviour, giving more flexibility to prove that all deadlines will be met. In [Poplavko et al. 2007], a tight estimate of worst-case performance is obtained by analysing all transitions from a scenario s to a scenario t. However, when the number of scenarios is large the amount of analysis required grows, as well as the overhead in the data stream. In our approach it is also possible, perhaps at the expense of less tight bounds on the worst-case execution times, to restrict the analysis to one per scenario (as demonstrated in the case-study, where the result turns out to be equally tight). Traditional SDF analysis often focusses on long-run behaviour such as average throughput. When considering scenario changes between SDF graphs however, an SDF graph executes for only a finite amount of time and the transient behaviour is of more interest than long-run average behaviour. [Moreira and Bekooij 2007] analyses transient behaviour of SDF graphs in order to study strictly periodic execution of graphs and latency of graphs. The paper shows that any valid static periodic schedule of the graph is an upper bound on the self-timed behaviour. We use similar linear models to bound transient behaviour, but we obtain the linear models in a different way. The paper does not study changes between different graphs.

The origin of our model lies in the Scenario Aware Data-Flow (SADF) model of [Theelen et al. 2006]. SADF also considers synchronous data-flow actors in different scenarios. It models the possible order in which scenarios occur stochastically by means of Markov models. Different scenarios may have different execution times

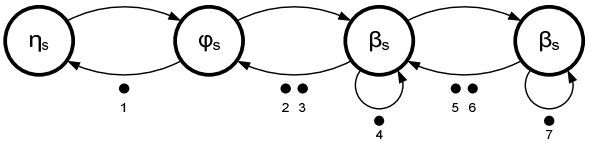

Fig. 2. HSDF model of MPEG-4 AVC codec

or even execution time distributions per actor, may have different communication rates and actors can be inactive in one or more scenarios. SADFs can be analysed stochastically and for worst-case behaviour. Long run average throughput can be computed, but also statistics of buffer occupancy. Exact results can in principle be obtained by state-space exploration techniques or simulation can be used to obtain approximations when the models become too large for exact analysis. Because the analysis is not done compositionally, per scenario as in this paper, the state-space that needs to be analysed can occasionally become prohibitively large. The techniques in this paper apply to the class of SADFs which are strongly consistent ([Theelen et al. 2006]) and which have fixed execution times instead of distributions, giving worst-case performance results, not using the scenario transition probabilities. This subclass of SADF is comparable to the HDF model of Girault, Lee and Lee. The analysis approach introduced in this paper is done on a per scenario basis and therefore avoids the state-space explosion problem associated with the monolithic analysis of the state-space of the combined scenarios.

3. PRELIMINARIES

3.1 Synchronous Data-Flow Graphs

In this paper, we study scenarios of Synchronous Data-Flow Graphs (SDFGs) [Lee and Messerschmitt 1987; Sriram and Bhattacharyya 2000]. An SDFG is a graph consisting of *actors* and *channels*, which *model* the execution of a real data-flow application (on a real platform). Figure 2 shows an example of an SDFG. It is taken from [Poplavko et al. 2007] and models a small part of an MPEG-4 Advanced Video Coding decoder. Actors (circles in the figure) represent the individual computations of an application. An actor can repeatedly *fire*, possibly multiple times simultaneously. With every firing of an actor, it consumes and produces fixed amounts of *tokens* on the FIFO channels. It can only fire if sufficient tokens are available on the channels from which it consumes. The production and consumption rates are indicated next to the channel ends. For readability this is omitted in cases where the rate is equal to 1. In Figure 2 all rates are equal to 1; such an SDFG is called *homogeneous*, an HSDFG. Tokens thus capture dependencies between actor firings. Such dependencies may originate from data dependencies, but also from dependencies on shared resources. Actors do not always represent actual computations, but may also be used to model communication (actors in Figure 2 model communication over a network-on-chip [Poplavko et al. 2007]), or synchronisation, for instance to express more complicated schedules as we illustrate in Section 7. For a more detailed, formal, semantics of SDF and its properties we refer to [Lee and Messerschmitt 1987; Sriram and Bhattacharyya 2000; Ghamarian et al. 2006].

Firings of actors take time. In the SDF model, actor firings are modelled with a

constant execution time, which is typically an upper bound on the real execution time of the corresponding computation of the actual application. 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 graph.

Because of the constant rates with which tokens are produced or consumed, SDFGs are known to execute in fixed repetitive patterns, called *iterations*. Such 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 a frame in an audio or video stream. This makes iterations the natural granularity for defining scenarios, both from the application point of view as from the performance analysis point of view. Note that this does not preclude that the execution of subsequent iterations may overlap in time. In fact, this is quite common in pipelined execution. Hence, different scenarios may be active simultaneously.

3.2 Max-Plus Algebra

We use the theory of *Max-Plus algebra* [Baccelli et al. 1992] as the semantical framework to model our scenarios of SDF graphs. Two essential operations that determine the self-timed execution of an SDFG are synchronisation and delay. Synchronisation takes place when an actor waits for all of its input tokens to become available and it fires immediately when all required tokens are present. There is a delay between the start of the firing and the end of the firing, when all the output tokens are produced, namely the execution time of the actor. These two elements fit very well with the Max-Plus 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 that 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-Plus expression.

The behaviour of a data-flow graph can be characterised in a dual manner, either by the times at which actors start their firings, or by the times at which the tokens in channels are produced. Both ways can be derived from each other. In this paper we focus on token production times, because the tokens capture exactly the dependencies between iterations that are crucial to studying scenario transition behaviour. We consider the collection of tokens that exist in their various channels in between iterations. The production times of these tokens are collected in a vector $\overline{\gamma}$ consisting of as many entries as there are tokens initially in the graph (e.g., seven in Figure 2). After one iteration, by definition [Lee and Messerschmitt 1987], there

exist the same number of tokens in the same channels, but with different times at which they are produced; $\overline{\gamma}_0$ becomes after an iteration of the graph $\overline{\gamma}_1$. We refer to such a vector as a *schedule* and use $\overline{\gamma}_k$ to denote the vector of production times of the tokens after the k-th iteration. The process of execution of an SDFG can be captured by means of a matrix-vector multiplication in Max-Plus, this is explained in more detail in the next section. 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.

$$\overline{\gamma}_{k+1} = \mathbf{G}\overline{\gamma}_k$$

We introduce some notation related to Max-Plus algebra. Max-Plus algebra defines the operations of the maximum of numbers and addition over the set $\mathbb{R} \cup \{-\infty\}$, 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 often used in Max-Plus literature. For scalars x and y, $x \cdot y$ (with shorthand xy) denotes ordinary multiplication, not the Max-Plus $\otimes$ 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-Plus 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. Unless stated otherwise, vectors are column vectors. To save space we sometimes write a row vector using transposition ($\overline{\boldsymbol{a}}^T$) to turn it into a column vector. In this paper we only use Max-Plus matrix-vector multiplication and never the traditional matrix-vector multiplication. For a matrix $\mathbf{M}$ and vector $\overline{\boldsymbol{x}}$, we use $\mathbf{M}\overline{\boldsymbol{x}}$ to denote this Max-Plus matrix multiplication. If $\overline{\boldsymbol{a}} = [a_i]$ and $\overline{\boldsymbol{b}} = [b_i]$ with $a_i, b_i \in \mathbb{R} \cup \{-\infty\}$ are vectors of size k, then we write $\overline{\boldsymbol{a}} \preceq \overline{\boldsymbol{b}}$ to denote that for every $1 \leq i \leq k$, $a_i \leq b_i$. With $\overline{\boldsymbol{a}}$ a vector and c a scalar, we use $c + \overline{\boldsymbol{a}}$ or $\overline{\boldsymbol{a}} + c$ to denote a vector with entries identical to the entries of $\overline{\boldsymbol{a}}$ with c added to each of them: $c + \overline{\boldsymbol{a}} = \overline{\boldsymbol{a}} + c = [a_i + c]$. We use $\overline{\mathbf{0}}$ to denote a vector with all zero-valued entries. The size of $\overline{\mathbf{0}}$ is clear from the context. We use $\max(\overline{\boldsymbol{a}}, \overline{\boldsymbol{b}})$ (resp. $\min(\overline{\boldsymbol{a}}, \overline{\boldsymbol{b}})$), defined as $[\max(a_i, b_i)]$ (resp. $[\min(a_i, b_i)]$) as a max (resp. min) operator on vectors and $\overline{\boldsymbol{a}} + \overline{\boldsymbol{b}}$ (resp. $\overline{\boldsymbol{a}} - \overline{\boldsymbol{b}}$), defined as $[a_i + b_i]$ (resp. $[a_i - b_i]$) as addition (resp. subtraction) of vectors. (Just like in traditional algebra subtraction is defined as addition of the additive inverse.) $||\overline{\boldsymbol{a}}||$ denotes a vector norm, defined as: $||\overline{\boldsymbol{a}}|| = \max_i a_i$. It is a vector norm in Max-Plus algebra, because (i) $||\overline{\boldsymbol{a}}|| = -\infty$ iff $a_i = -\infty$ for all i; (ii) $||c + \overline{\boldsymbol{a}}|| = c + ||\overline{\boldsymbol{a}}||$; (iii) $||\max(\overline{\boldsymbol{a}}, \overline{\boldsymbol{b}})|| \leq \max(||\overline{\boldsymbol{a}}||, ||\overline{\boldsymbol{b}}||)$. For a vector $\overline{\boldsymbol{a}}$ with $||\overline{\boldsymbol{a}}|| > -\infty$, we use $\overline{\boldsymbol{a}}^{norm}$ to denote $\overline{\boldsymbol{a}} - ||\overline{\boldsymbol{a}}||$, the normalised vector $\overline{\boldsymbol{a}}$, so that $||\overline{\boldsymbol{a}}^{norm}|| = 0$. An inner product is defined as follows: $(\overline{\boldsymbol{a}}, \overline{\boldsymbol{b}}) := \max_i a_i + b_i$. If matrix $\mathbf{M} = [\overline{\boldsymbol{m}}_j]$ (i.e., has column vectors $\overline{\boldsymbol{m}}_j$) then $\mathbf{M}\overline{\boldsymbol{x}} := \max_j(\overline{\boldsymbol{m}}_j + \overline{\boldsymbol{x}})$ and $\mathbf{M}^T\overline{\boldsymbol{x}} := [(\overline{\boldsymbol{m}}_j, \overline{\boldsymbol{x}})]$. It is easy to verify that matrix multiplication is linear: $\mathbf{M}(\max(\overline{\boldsymbol{x}}, \overline{\boldsymbol{y}})) = \max(\mathbf{M}\overline{\boldsymbol{x}}, \mathbf{M}\overline{\boldsymbol{y}})$ and $\mathbf{M}(c + \overline{\boldsymbol{x}}) = c + \mathbf{M}\overline{\boldsymbol{x}}$. If $\mathbf{M} = [\overline{\boldsymbol{m}}_j]$ and $\mathbf{N} = [\overline{\boldsymbol{n}}_j]$ are equally sized matrices with k columns, then $\mathbf{M} \preceq \mathbf{N}$ denotes that for all $1 \leq j \leq k$, $\overline{\boldsymbol{m}}_j \preceq \overline{\boldsymbol{n}}_j$.

4. MODELLING SDF WITH MAX-PLUS

In this section we discuss how Max-Plus linear algebra can be used to model the behaviour of an SDFG. In particular, we give an alternative operational semantics

to the traditional one (see e.g. [Ghamarian et al. 2006]). The main advantage of this semantic model is that individual iterations of the graph can be easily separated, despite the fact that they may overlap in time when they are executed in a pipelined fashion. This is an important advantage when dealing with scenarios, because they are associated with iterations of the graphs and can be pipelined. In this framework they can be easily handled individually.

In the classical operational semantics of SDF [Ghamarian et al. 2006], the global evolution of the state of the SDFG is modelled by a labelled transition system. Every state in this transition system represents the graph's condition at a particular point in time. In contrast, in the Max-Plus semantics we focus on determining the appropriate time-stamps for every token produced and consumed in the graph. We group those tokens together that belong to the same iteration of the graph instead of those tokens that are alive at the same point in time.

4.1 Example

We illustrate this using the graph of Figure 3. The repetition vector of the graph is $[1;\ 2;\ 2;\ 1]^T$, meaning that one iteration of the graphs consists of one firing of actors A and D each and two firings of actors B and C each. The self-timed execution of this graph is illustrated in Figure 4. Time is depicted on the horizontal axis and the actors vertically. Grey boxes represent the firings of the actors, labelled by the actor name, the number of the iteration and for actors B and C the number of the firing within that iteration. The small circles denote the initial tokens and the tokens produced at the end of every iteration.

If we assume that, initially, all tokens have a time-stamp of 0 (are available, starting from time 0), then, actor A fires at time 0. The firing takes 2 units of time and A produces 2 tokens to actor B, 2 tokens to actor C and 1 token on its self-edge, each with a time-stamp of 2. Actors B and C depend on these tokens and can now start at time 2. In fact, two firings of C can start simultaneously at time 2, but only one firing of B, because it has a self-edge with one token only. B takes 4 units of time and ends its firing at time 6. This produces a token labelled with time-stamp 6 on its self-edge and on the edge to actor D. A second firing of B is started consuming the token from the self edge and the token received earlier from A. Meanwhile at time 8, the two firings of C end and the second firing of B ends at 10. D gets two tokens from actor B, time-stamped 6 and 10 respectively and two tokens from C, both time-stamped 8. Hence, the firing of D starts at 10 and ends at 12. This completes one iteration. The tokens are back on their original positions and the time-stamps are 2 for t_1, 10 for t_2, 0 for t_3 and 12 for t_4. These are the tokens labelled '1' in Figure 4. We use these time-stamps to represent the 'state' of the graph after one iteration. Note the difference with the states in the traditional operational semantics [Ghamarian et al. 2006] which represent a global state of the graph at a distinguished point in time (a vertical cut in the graph of Figure 4). In such a state however, typically different iterations are active simultaneously.

We represent the time-stamps of the four initial tokens with a vector using the order $[t_1;\ t_2;\ t_3;\ t_4]^T$ as indicated in Figure 3. The initial vector is $\overline{\gamma}_0 = [0;\ 0;\ 0;\ 0]^T$. After one iteration this becomes $\overline{\gamma}_1 = [2;\ 10;\ 0;\ 12]^T$, after two iterations $\overline{\gamma}_2 = [4;\ 18;\ 12;\ 20]^T$ and so on: $\overline{\gamma}_3 = [14;\ 26;\ 20;\ 28]^T$ and $\overline{\gamma}_4 = [22;\ 34;\ 28;\ 36]^T$. $\overline{\gamma}_4$ is equal to $\overline{\gamma}_3$, except that all time-stamps have been shifted by 8 time units. From

this fact it is easy to see that all subsequent iterations will produce again similar vectors, each time shifted by 8 time units; the graph executes with a period of 8 or a throughput of 1/8.

In order to derive a Max-Plus equation for the graph behaviour, we study this model of execution symbolically. We show that at any point in time, the time-stamp of a token can be expressed symbolically, as a formula, a Max-Plus linear combination, as $t = \max_i t_i + g_i$ for suitable constants g_i, where the t_i are the time-stamps of the initial tokens. This can be written in the form of a Max-Plus vector inner-product $(\overline{\boldsymbol{g}}, \overline{\boldsymbol{t}})$ for a corresponding vector $\overline{\boldsymbol{g}} = [g_1, g_2, g_3, g_4]^T$ and with $\overline{\boldsymbol{t}} = [t_1, t_2, t_3, t_4]^T$. We refer to such an equation as a *symbolic time-stamp* and symbolic time-stamps will be represented by the vector $\overline{\boldsymbol{g}}$. As an example we consider again the execution of the SDFG of Figure 3, this time with symbolic time-stamps. Time-stamp t_1 corresponds to the symbolic time-stamp vector $[0;\ -\infty;\ -\infty;\ -\infty]^T$, t_2 corresponds to the vector $[-\infty;\ 0;\ -\infty;\ -\infty]^T$, t_3 to $[-\infty;\ -\infty;\ 0;\ -\infty]^T$ and finally t_4 to $[-\infty;\ -\infty;\ -\infty;\ 0]^T$. We start by firing actor A consuming two tokens, one from the self edge and one from the edge from actor D, labelled with $[0;\ -\infty;\ -\infty;\ -\infty]^T$ and $[-\infty;\ -\infty;\ 0;\ -\infty]^T$ respectively. The tokens produced by A carry the symbolic time-stamp:

$$\max([0;\ -\infty;\ -\infty;\ -\infty]^T, [-\infty;\ -\infty;\ 0;\ -\infty]^T) + 2 = [2;\ -\infty;\ 2;\ -\infty]^T$$

which corresponds to the expression $\max t_1 + 2, t_3 + 2$. The subsequent firing of actor B with a duration of 4 consumes one of these tokens as well as an initial token on its self-edge labelled with $[-\infty;\ 0;\ -\infty;\ -\infty]^T$. B produces output tokens labelled as:

$$\max([2;\ -\infty;\ 2;\ -\infty]^T, [-\infty;\ 0;\ -\infty;\ -\infty]^T) + 4 = [6;\ 4;\ 6;\ -\infty]^T$$

If we continue the symbolic execution till the completion of the iteration, we obtain four new tokens t'_1, t'_2, t'_3 and t'_4 with respectively the symbolic time-stamps $[2;\ -\infty;\ 2;\ -\infty]^T$, $[10;\ 8;\ 10;\ -\infty]^T$, $[-\infty;\ -\infty;\ -\infty;\ 0]^T$ and $[12;\ 10;\ 12;\ -\infty]^T$. If we collect the symbolic time stamps of these new tokens into a new vector $[t'_1;\ t'_2;\ t'_3;\ t'_4]^T$, we obtain the following Max-Plus matrix equation:

$$\begin{bmatrix} t'_1 \\ t'_2 \\ t'_3 \\ t'_4 \end{bmatrix} = \begin{bmatrix} 2 & -\infty & 2 & -\infty \\ 10 & 8 & 10 & -\infty \\ -\infty & -\infty & -\infty & 0 \\ 12 & 10 & 12 & -\infty \end{bmatrix} \begin{bmatrix} t_1 \\ t_2 \\ t_3 \\ t_4 \end{bmatrix}$$

In short:

$$\overline{\boldsymbol{t}}' = \mathbf{G}\overline{\boldsymbol{t}}.$$

In this way, the behaviour of any SDFG F can be characterised by a corresponding Max-Plus matrix $\mathbf{G}_F$. This matrix can be computed by means of a symbolic execution of one iteration of the graph as illustrated for the example. Algorithm 1 gives a sketch of an algorithm to compute the matrix for an SDFG. The notation and definitions of SDF graph are left somewhat implicit, because they will not be needed for the remainder of the paper.

Because there is a straightforward translation from a concrete Max-Plus matrix to a corresponding *homogeneous* SDFG, this algorithm also provides an alternative

translation from an SDFG to an HSDFG. In the traditional translation of [Sriram and Bhattacharyya 2000], the resulting HSDF has a separate actor for every individual actor firing in an iteration of the original graph and the firing time of those corresponding actors exactly coincide. Therefore, the size of the resulting HSDFG for the traditional translation [Sriram and Bhattacharyya 2000] equals the sum of the entries on the repetition vector of the graph. If we use a translation from the corresponding Max-Plus matrix to an HSDFG, the translation results in a graph with a number of actors that grows in the worst case with the square of the number of initial tokens in the graph (but can be much smaller if the matrix is sparse). For some graphs (in particular graphs with few initial tokens) this may result in a much smaller HSDFG. This is possible because the equivalence relation maintained between the original graph and its HSDF conversion is slightly weaker than for the traditional translation. There are no corresponding actor firings as in the translation of [Sriram and Bhattacharyya 2000], but the initial tokens after every iteration of the graph are produced with exactly the same time-stamps and hence the graph has, for instance, the same throughput. Details of the translation algorithm can be found in [Geilen 2009].

4.2 Max-Plus model of self-timed execution

The matrix equation relates the time-stamps of the tokens before and after one iteration under self-timed execution. Therefore, the self-timed evolution of an SDFG, starting from initial tokens with time stamps $\overline{\gamma}_0$ is entirely captured by the sequence

$$\{\overline{\gamma}_k \mid k \geq 0\} = \{\mathbf{G}^k\overline{\gamma}_0 \mid k \geq 0\}.$$

It is known [Baccelli et al. 1992] that the sequence $\mathbf{G}^k\overline{\gamma}_0$ ultimately becomes periodic for proper (connected, self-timed bounded [Ghamarian et al. 2006]) graphs. I.e., there exist $m, n \in \mathbb{N}$ such that for every $k \geq m$, $\mathbf{G}^{k+n}\overline{\gamma}_0 = \mathbf{G}^k\overline{\gamma}_0 + n\lambda$, where λ is the maximum cycle mean (MCM) of the equivalent homogeneous SDFG [Sriram and Bhattacharyya 2000; Ghamarian et al. 2006].

Algorithm 2 implements an exploration of the behaviour of the graph in a recursive algorithm. MinPeriod($\mathbf{G}, \overline{\gamma} = \overline{\mathbf{0}}, k = 0, S = \varnothing$) takes as arguments a matrix $\mathbf{G}$ of the SDFG, an initial schedule $\overline{\gamma}$ (with default value zero vector $\overline{\mathbf{0}}$), a number k, the number of iterations already explored (default 0) and a set S of normalised vectors already observed in the exploration (initially $\varnothing$). The algorithm explores subsequent iterations of the graph until the periodic phase of the execution is detected (condition line 2). It then returns the MCM λ and the first vector in the periodic phase (line 3), else it continues exploration recursively (line 5). The algorithm is initially called with only the matrix $\mathbf{G}$, MinPeriod($\mathbf{G}$).

Note that although we focus on production times of tokens, (models of) actor firing times can be derived from (the models of) the production times of the tokens. For a given actor a, we can find a matrix $\mathbf{A}_a$ such that $\mathbf{A}_a\overline{\gamma}_k$ is a vector containing all the firing times of actor a in iteration k.

One of the essential properties of SDFGs is that their temporal behaviour is *monotonic*:

$$\overline{\gamma}_a \preceq \overline{\gamma}_b \Rightarrow \mathbf{G}\overline{\gamma}_a \preceq \mathbf{G}\overline{\gamma}_b$$

Suppose $\overline{\gamma}_a$ represents the real moments at which the tokens become available and

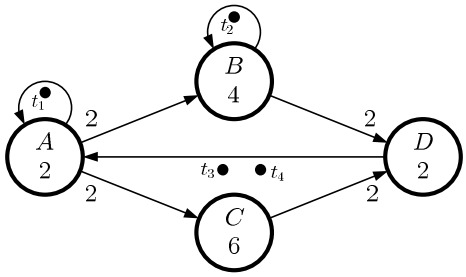

Fig. 3. Example SDFG

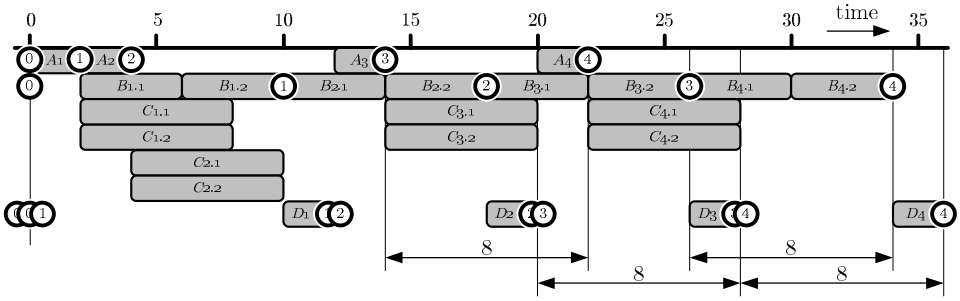

Fig. 4. Self-timed execution of the example SDFG

$\overline{\gamma}_b$ is an upper bound we manage to establish, then the equation gives us an upper bound on the production times of the new generation of tokens, by assuming $\overline{\gamma}_b$ were the actual production times. In fact, this is also true for the entries in the matrix $\mathbf{G}$, i.e. for the actor execution times, (constant) upper bounds are used instead of the real (varying) execution times. Such bounds are obtained for instance by means of worst-case execution time analysis.

$$\mathbf{G}_a \preceq \mathbf{G}_b \Rightarrow \mathbf{G}_a\overline{\boldsymbol{\gamma}} \preceq \mathbf{G}_b\overline{\boldsymbol{\gamma}}$$

Moreover, a bound can be established on this difference:

$$||\mathbf{G}\overline{\boldsymbol{\gamma}}_b - \mathbf{G}\overline{\boldsymbol{\gamma}}_a|| \leq ||\overline{\boldsymbol{\gamma}}_b - \overline{\boldsymbol{\gamma}}_a||$$

respectively

$$||\mathbf{G}_a\overline{\boldsymbol{\gamma}} - \mathbf{G}_b\overline{\boldsymbol{\gamma}}|| \leq ||\mathbf{G}_a - \mathbf{G}_b||$$

which shows that the difference does not further increase.

4.3 Spectral Analysis

In the tradition of linear system theory, to study this evolution $\overline{\boldsymbol{\gamma}}_k = \mathbf{G}^k\overline{\boldsymbol{\gamma}}_0$ in the long run, *spectral analysis* of the matrix $\mathbf{G}$ is used. Spectral analysis deals with the study of the *eigenvalue equation*, which is in Max-Plus terms:

$$\mathbf{G}\overline{\boldsymbol{\gamma}} = \lambda + \overline{\boldsymbol{\gamma}}.$$

For any combination of λ and $\overline{\boldsymbol{\gamma}}$ with $||\overline{\boldsymbol{\gamma}}|| > -\infty$ that satisfy this equation, λ is called the *eigenvalue* and $\overline{\boldsymbol{\gamma}}$ the corresponding *eigenvector*. It is known [Baccelli

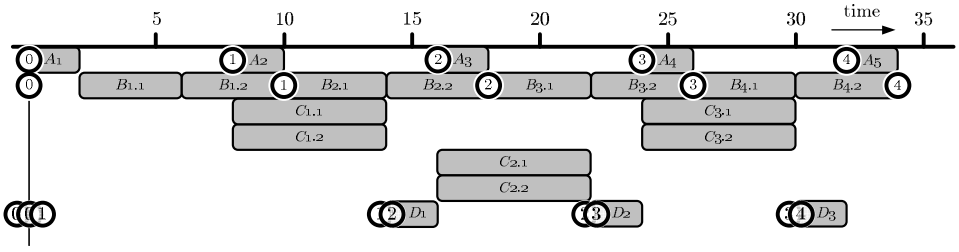

Fig. 5. Alap execution of the example SDFG

Algorithm 1 Compute Max-Plus matrix of an SDFG

1: ComputeMatrix(F)
2: /* T associates symbolic time-stamp $\bar{\boldsymbol{i}}_k$ to initial token t_k */
3: $T \leftarrow \left\{(t_k, \bar{\boldsymbol{i}}_k) \mid t_k \in InitialTokens\right\}$
4: $\sigma \leftarrow$ SeqSchedule(F) /* determine a sequential schedule of an iteration */
5: **for** $j = 1$ to Length(σ) **do**
6: Actor $a \leftarrow \sigma[j]$ /* a the j'th actor in the schedule */
7: Fire a consuming tokens $U \subseteq T$
8: produce output tokens with time-stamp $\overline{\boldsymbol{g}}_p \leftarrow \max\{\overline{\boldsymbol{g}}(t) \mid t \in U\} + E(a)$
9: add new tokens to T
10: **end for**
11: $\mathbf{G} \leftarrow [\overline{\boldsymbol{g}}(t)]$ for all $t \in InitialTokens$

Algorithm 2 Algorithm to compute minimal period (throughput) of an SDFG and schedule in the periodic phase

1: MinPeriod($\mathbf{G}, \overline{\gamma} = \overline{\mathbf{0}}, k = 0, S = \varnothing$)
2: **if** $(\overline{\gamma}_s, m) \in S$ with ${\overline{\gamma}_s}^{norm} = \overline{\gamma}^{norm}$ **then**
3: **return** $(||\overline{\gamma} - \overline{\gamma}_s||/(k-m), \overline{\gamma})$
4: **else**
5: **return** MinPeriod($\mathbf{G}, \mathbf{G}\overline{\gamma},\ k+1,\ S \cup \{(\overline{\gamma}, k)\}$)
6: **end if**

et al. 1992; Ghamarian et al. 2006] that any connected and self-timed bounded SDFG has a corresponding matrix with a unique eigenvalue . If $\overline{\gamma}$ is an eigenvector, then so is $c+\overline{\gamma}$ for any c. The eigenvalue is identical to the MCM of the equivalent homogeneous SDFG [Karp and Miller 1966; Sriram and Bhattacharyya 2000], and thus indicates the maximum achievable throughput ($1/\lambda$) of the graph. We observed that for the graph of Figure 3, $\overline{\gamma}_4 = \mathbf{G}\overline{\gamma}_3 = \overline{\gamma}_3 + 8$. This shows that $\overline{\gamma}_3$ (and thus also $\overline{\gamma}_4$) is an eigenvector of $\mathbf{G}$ with eigenvalue 8. In normalised form they are equal to $[-14;\ -2;\ -8;\ 0]$.

When a graph executes a prolonged time in a self-timed way within one scenario, the behaviour converges to steady state behaviour. The graph then executes at its maximal throughput and in a periodic way. It is known that this periodic behaviour may span multiple iterations [Baccelli et al. 1992], but the length of the period is exactly one iteration when the vector $\overline{\gamma}$ is an eigenvector of the matrix $\mathbf{G}$ (and in

Algorithm 3 Compute eigenvector and eigenvalue

1: SpectralAnalysis($\mathbf{G}$)
2: $(\lambda, \overline{\gamma}_p) \leftarrow$ MinPeriod ($\mathbf{G}$)
3: $\overline{\gamma} \leftarrow \overline{\gamma}_p, \quad \overline{\gamma}^S \leftarrow \overline{\gamma}_p$
4: **repeat**
5: $\quad \overline{\gamma}^S \leftarrow \max(\overline{\gamma}^S, \overline{\gamma})$
6: $\quad \overline{\gamma} \leftarrow \mathbf{G}\overline{\gamma} - \lambda$
7: **until** $\overline{\gamma} = \overline{\gamma}_p$
8: **return** $(\overline{\gamma}^S - ||\overline{\gamma}^S||, \lambda)$

that case, the duration of the period is the eigenvalue of the matrix). Computing eigenvector and eigenvalue is called *spectral analysis* in linear system theory. For SDFGs it can be done with Algorithm 3, which is based on the following proposition.

Proposition 4.1. *Let* $\mathbf{G}$ *be a Max-Plus matrix of an SDFG with eigenvalue* λ. *Let* $\{\overline{\gamma}_k \mid 1 \leq k \leq n\}$ *be the periodic phase in the self-timed execution of the graph, i.e.,* $\overline{\gamma}_{k+1} = \mathbf{G}\overline{\gamma}_k$ *and* $\mathbf{G}\overline{\gamma}_n = \overline{\gamma}_1 + n\lambda$. *Then* $\overline{\gamma}^S$, *defined as*

$$\overline{\gamma}^S = \max_{1 \leq k \leq n} (\overline{\gamma}_k - k\lambda)$$

is an eigenvector of $\mathbf{G}$.

Proof. (Following [Heidergott et al. 2006], Theorem 4.1.) We show that $\overline{\gamma}^S$ satisfies the eigenvalue equation with eigenvalue λ, using the fact that matrix multiplication is linear and that because of the periodicity, $\overline{\gamma}_{n+1} = \overline{\gamma}_1 + n\lambda$.

$$\begin{aligned}
\mathbf{G}\overline{\gamma}^S &= \mathbf{G}(\max_{1 \leq k \leq n} \overline{\gamma}_k - k\lambda) \\
&= \max_{1 \leq k \leq n} \mathbf{G}(\overline{\gamma}_k - k\lambda) \\
&= \max_{1 \leq k \leq n}(\mathbf{G}\overline{\gamma}_k) - k\lambda \\
&= \max_{1 \leq k \leq n} \overline{\gamma}_{k+1} - k\lambda \\
&= \lambda + \max_{1 \leq k \leq n} \overline{\gamma}_{k+1} - (k+1)\lambda \\
&= \lambda + \max_{1 \leq k \leq n} \overline{\gamma}_k - k\lambda \\
&= \lambda + \overline{\gamma}^S
\end{aligned}$$

□

It follows from the proposition that an eigenvector can be computed by taking the maximum over all vectors encountered in the periodic phase of the self-timed execution shifted by a corresponding factor $k\lambda$. The algorithm invokes the throughput calculation algorithm of Algorithm 2, which returns the MCM (minimum period, eigenvalue, the reciprocal of the throughput) of the graph, as well as the first vector in the periodic phase of the execution. The algorithm then executes through the periodic phase once, while maintaining the max over all vectors encountered in $\overline{\gamma}^S$. This is an eigenvector according to Proposition 4.1. It is returned by the algorithm (normalised), together with the eigenvalue obtained from the invocation of Algorithm 2.

Alternative to Algorithm 3, traditional Max-Plus spectral analysis algorithms can be used [Cochet-Terransson et al. 1999; Dasdan et al. 1999], but the algorithm presented here can work directly on the SDF graph. Note further that we assume

that time stamps in schedules can be arbitrary real numbers. In real world systems often, actors can only be scheduled on discrete (integer) moments in time. The existence of an eigenvector depends on this assumption. However, it is easy to show [Chao and Sha 1995] that a periodic schedule with approximately the same performance can be obtained by rounding the schedule or an optimal schedule by unfolding the graph.

An eigenvector is a vector which repeats itself with every iteration under *as-soon-as-possible (asap, self-timed)* scheduling. There is a similar schedule which is periodic under *as-late-as-possible (alap)* scheduling. Alap scheduling of an SDF graph can be mimicked by reversing all dependency edges of the graph and considering an asap scheduling of the resulting graph, as if it were evolving backwards in time. In terms of Max-Plus, the alap schedule $\overline{\gamma}^L$ of a matrix is the inverse $-\overline{\gamma}'$ of the eigenvector $\overline{\gamma}'$ of the transposed matrix $\mathbf{G}^T$. For example, for the graph of Figure 3, the asap eigenvector equals $[-14;\ -2;\ -8;\ 0]^T$ and the alap eigenvector equals $[-8;\ -6;\ -8;\ 0]^T$. If we align asap schedule $\overline{\gamma}^S$ and alap schedule $\overline{\gamma}^L$, by adding a suitable scalar, such that $||\overline{\gamma}^S - \overline{\gamma}^L|| = 0$, then $\overline{\gamma}^S(i) = \overline{\gamma}^L(i)$ exactly for the tokens on the critical path of execution. For the example this becomes $\overline{\gamma}^L = [-4;\ -2;\ -4;\ 4]^T$. The token on the self-edge of actor B is critical, which explains that the minimum period of the graph is 8, two firings of B are required, each taking 4 units of time. $\overline{\gamma}^L - \overline{\gamma}^S = [10;\ 0;\ 4;\ 4]^T$, showing the slack in the firing of the other actors. In general, $\overline{\gamma}^L(j) - \overline{\gamma}^S(j)$ is the amount of slack for token j. Figure 5 show the alap execution of the graph of Figure 3. The firings of actor B are on the critical path and have been kept at the same position. Other actors are fired as late as possible to enable these firings to take place. The alap tokens are positioned right before they are consumed.

A schedule bounded by the alap schedule can be executed within a period of the eigenvalue as proved in the following proposition.

PROPOSITION 4.2. *Let* $\mathbf{G}$ *be a Max-Plus matrix of an SDFG. Let* $-\overline{\gamma}^L$ *be an eigenvector of* $\mathbf{G}^T$ *(the periodic alap schedule) with corresponding eigenvalue* λ *(also the eigenvalue of* $\mathbf{G}$*), then* $\mathbf{G}\overline{\gamma}^L \preceq \overline{\gamma}^L + \lambda$.

PROOF. It is easy to show that $\mathbf{G}^T$ and $\mathbf{G}$ have the same eigenvalue. Let $\mathbf{G} = [g_{km}]$. From $\mathbf{G}^T(-\overline{\gamma}^L) = \lambda - \overline{\gamma}^L$ it follows that for all m, $\max_k g_{km} - \overline{\gamma}^L(k) = \lambda - \overline{\gamma}^L(m)$. This implies that for all m and k, $g_{km} + \overline{\gamma}^L(m) \leq \lambda + \overline{\gamma}^L(k)$. For all k, $\max_m g_{km} + \overline{\gamma}^L(m) \leq \lambda + \overline{\gamma}^L(k)$. Since the left-hand side equals $(\mathbf{G}\overline{\gamma}^L)(k)$, we have that for all k, $(\mathbf{G}\overline{\gamma}^L)(k) \leq \lambda + \overline{\gamma}^L(k)$ and thus that $\mathbf{G}\overline{\gamma}^L \preceq \lambda + \overline{\gamma}^L$. □

A straightforward corollary of this proposition and monotonicity of Max-Plus and SDF behaviour is that

$$\overline{\gamma} \preceq \overline{\gamma}^L \quad \Rightarrow \quad \mathbf{G}\overline{\gamma} \preceq \overline{\gamma}^L + \lambda$$

end hence by repeating this property

$$\overline{\gamma} \preceq \overline{\gamma}^L \quad \Rightarrow \quad \mathbf{G}^k\overline{\gamma} \preceq \overline{\gamma}^L + k\lambda$$

5. ANALYSING SCENARIOS

In this paper, we consider scenario changes to modify the graph and hence the matrix $\mathbf{G}$ in between iterations and therefore the analysis is not focussed on the

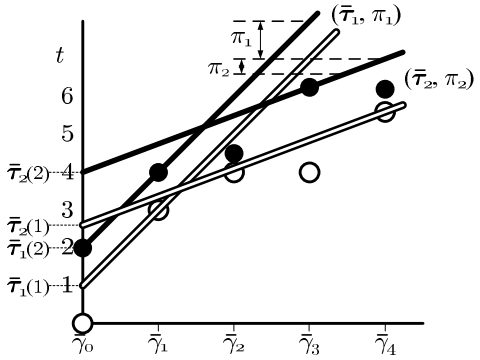

Fig. 6. Different linear upper bounds

long-run average behaviour of the graph (only), but on transient behaviour between scenarios. In this case, the final $\overline{\gamma}_k$ of one scenario provides the initial conditions for the execution of the next. We model a Scenario Aware Data Flow graph $(\Sigma, \{\mathbf{G}_s \mid s \in \Sigma\})$ as a set Σ of all scenarios and corresponding SDF graphs modelled by matrices $\mathbf{G}_s$.

For instance, for the MPEG-4 Simple Profile decoder of Figure 1 different scenarios correspond to different numbers of macro blocks and different execution times for I or P frames. Every individual scenario is captured by a specific SDF graph, where x in the graph is replaced by a concrete rate and the execution times of the actors have a concrete value.

5.1 A Model of Transient Behaviour

To be able to effectively reason about the execution of the graphs in different scenarios we need a simple model of transient behaviour. We characterise the behaviour of an SDF graph, i.e., the sequence of vectors $\overline{\gamma}_k$, by a *linear upper bound*. (Note that it is straightforward to also include other tokens produced and consumed during the iterations, or actor firings if these are important for performance requirements.) We look for a vector $\overline{\boldsymbol{\tau}}$ and a scalar number π such that the following is valid for the iterations (k) of interest.

$$\overline{\gamma}_k \preceq \overline{\boldsymbol{\tau}} + k \cdot \pi \tag{1}$$

This linear upper bound, which we denote by the pair $(\overline{\boldsymbol{\tau}}, \pi)$, consists of two components,an initial delay vector $\overline{\boldsymbol{\tau}}$, and a period π, which determines the guaranteed throughput $(1/\pi)$.

Such bounds are illustrated in Figure 6. The figure shows a sequence of vectors $\overline{\gamma}_k$ along the horizontal axis. The vectors consist of two elements, represented by the vertical pairs of open en closed circles giving their value on the vertical axis. The two pairs of lines with identical slope (one solid line, one open line per pair) both constitute upper bounds. π_i corresponds to the slope of the lines and $\overline{\boldsymbol{\tau}}_i$ is the vector of intersections with the vertical axis. The figure illustrates that different linear upper bounds (for instance $(\overline{\boldsymbol{\tau}}_1, \pi_1)$ and $(\overline{\boldsymbol{\tau}}_2, \pi_2)$) can be models of the same execution sequence, short delay (smaller $\overline{\boldsymbol{\tau}}$) can be traded for short period (smaller π). We prefer linear upper bounds $(\overline{\boldsymbol{\tau}}, \pi)$ that are *tight*, i.e., such that there exists

no other linear upper bound model $(\overline{\boldsymbol{\tau}}', \pi')$ for the set of vectors such that both $\overline{\boldsymbol{\tau}}' \preceq \overline{\boldsymbol{\tau}}$ and $\pi' \leq \pi$ (i.e., they are Pareto optimal).

The linear upper bound model is comparable to linear arrival curves in Network Calculus [Boudec and Thiran 2003] or Real-Time Calculus [Thiele et al. 2000] or latency-rate curves [Stiliadis and Varma 1998]. As such, these models of SDF behaviour can also be suitable interfaces to analysis techniques based on these models.

5.2 Analysing Delay and Period

In order to use the linear upper bound (delay-period) model to analyse the performance of applications, we need an algorithm to determine the delay-period model(s) for any given SDFG. For instance, we might like to compute for a given desired period π (the throughput requirement) of the graph and an initial condition $\overline{\boldsymbol{\gamma}}_0$, the minimal delay $\overline{\boldsymbol{\tau}}$ such that $(\overline{\boldsymbol{\tau}}, \pi)$ is a tight model of the behaviour of the graph starting from specific initial conditions $\overline{\boldsymbol{\gamma}}_0$. In this case, we assume that the desired throughput and hence the period π are fixed and given. We give a more general algorithm later on, but first discuss a method how we can compute such a model. We start from the basic Max-Plus based state-space exploration method used in Algorithm 2 [Ghamarian et al. 2006]. This method computes the vectors $\overline{\boldsymbol{\gamma}}_k$ one by one by means of executing the SDFG model. The execution proceeds until a recurrent state is detected, i.e., if for the latest $\overline{\boldsymbol{\gamma}}_k$, there is some $n < k$ and constant λ such that $\overline{\boldsymbol{\gamma}}_k = \overline{\boldsymbol{\gamma}}_n + (k-n)\lambda$. In that case, λ is the eigenvalue of $\mathbf{G}$ (the minimum period the graph can support) and the rest of the vectors will proceed periodically according to the recursive equation

$$\overline{\boldsymbol{\gamma}}_{k+m} = \overline{\boldsymbol{\gamma}}_{n+m} + (k-n)\lambda.$$

Because of the periodicity, further exploration is not necessary.

We adapt the Max-Plus throughput analysis method as follows. During the exploration we maintain the smallest vector $\overline{\boldsymbol{\tau}}$ such that $(\overline{\boldsymbol{\tau}}, \pi)$ is a tight model for all schedules up to k. For $k = 0$, the initial model $(\overline{\boldsymbol{\tau}}, \pi)$ has $\overline{\boldsymbol{\tau}} = \overline{\boldsymbol{\gamma}}_0$. For every following k, we maintain the guarantee that $\overline{\boldsymbol{\gamma}}_k \preceq \overline{\boldsymbol{\tau}} + k \cdot \pi$ or $\overline{\boldsymbol{\gamma}}_k - k \cdot \pi \preceq \overline{\boldsymbol{\tau}}$by adapting $\overline{\boldsymbol{\tau}}$ accordingly. The algorithm thus proceeds as follows:

$$\begin{cases} \overline{\boldsymbol{\tau}}_0 = \overline{\boldsymbol{\gamma}}_0 \\ \overline{\boldsymbol{\tau}}_k = \max(\overline{\boldsymbol{\tau}}_{k-1}, \overline{\boldsymbol{\gamma}}_k - k \cdot \pi) \end{cases}$$

Once the execution is periodic and the desired period π is achievable by the graph in the long run ($\pi \leq \lambda$), then also the delay vector $\overline{\boldsymbol{\tau}}_k$ will not change anymore (proved below) and the analysis can terminate. $(\overline{\boldsymbol{\tau}}_k, \pi)$ is the required tight model. If $\lambda > \pi$, then the graph does not have any model with such period π. The following proposition proves that after the first recurrent state, the delay vector does not need to be increased anymore.

PROPOSITION 5.1. *If $\overline{\boldsymbol{\gamma}}_l$ is a recurrent state in the state-space exploration and $\overline{\boldsymbol{\gamma}}_k \preceq \overline{\boldsymbol{\tau}} + k\pi$ for all $k < l$ and $\pi \geq \lambda$, then $\overline{\boldsymbol{\gamma}}_m \preceq \overline{\boldsymbol{\tau}} + m\pi$ for all $m \in \mathbb{N}$.*

PROOF. It is true for $m < l$ by the assumption. For $m \geq l$, there is some $k < m$ such that $\overline{\boldsymbol{\gamma}}_m = \overline{\boldsymbol{\gamma}}_k + (m-k)\lambda \preceq \overline{\boldsymbol{\gamma}}_k + (m-k)\pi \preceq \overline{\boldsymbol{\tau}} + k\pi + (m-k)\pi = \overline{\boldsymbol{\tau}} + m\pi$. □

We see in Section 6 that in the context of reconfigurations from one scenario to another, we may want to enforce a particular schedule as a reference and compute a single delay relative to that schedule. Let $\overline{\gamma}^*$ be that schedule and let our goal be to find the smallest delay τ such that for all k

$$\overline{\gamma}_k \preceq \tau + \overline{\gamma}^* + k \cdot \pi$$

Note that τ in this case is not a vector but a constant scalar. To find it, we can generalise the earlier problem to look for the smallest delay vector $\overline{\boldsymbol{\tau}}$ such that for all k

$$\overline{\gamma}_k \preceq \overline{\boldsymbol{\tau}} + \overline{\gamma}^* + k \cdot \pi \tag{2}$$

This is a generalisation of both problems. The delay we are looking for in the latter problem equals: $\tau = ||\overline{\boldsymbol{\tau}}||$ and it reduces to the former problem by taking $\overline{\gamma}^* = \overline{\mathbf{0}}$.

Before we give an algorithm, we generalise the problem a bit more. As discussed in the beginning of this section, the delay-period model of actor firings may allow for solutions consisting of different (tight) combinations of delay $\overline{\boldsymbol{\tau}}$ and period π. We find *all* combinations that provide a model and know all combinations that are tight. To do this, we observe the following. Instead of interpreting the upper bound equation (2) as an equation with constant $\overline{\boldsymbol{\tau}}$ and π and variables $\overline{\gamma}_k$ and k, we can also reverse the roles in an execution of the graph. Every observed point $k, \overline{\gamma}_k$ in the exploration of the state space limits the combinations of $\overline{\boldsymbol{\tau}}$ and π that are models by equation (2). In addition to that, once the periodic phase of the behaviour is reached, its eigenvalue is known and the periodic extension is captured by a lower bound of λ on π representing the minimal period or maximal achievable throughput. It follows from Proposition 5.1 that all equations up to that point are sufficient to guarantee that their solutions are models of the entire, infinite execution.

Note that the set of equations thus obtained consist of half spaces in a space consisting of one of the entries of $\overline{\boldsymbol{\tau}}$ and the period π. There are no constraints among $\overline{\boldsymbol{\tau}}$ entries directly. The total set of admissible combinations is given by the intersection of these half-spaces and the *tight* models are exactly the so-called *Pareto frontier* of this set.

The equations for the vectors of Figure 6 are illustrated in Figure 7. Concrete delay-period models are constrained such that the solid lines lie in the dark grey area and the open lines are in the are consisting of the light and the dark grey areas. The two lines drawn in the figure illustrate an example model. In this case the (unique) tight model having the minimal possible period. All tight models touch on the dashed lines at some point of the graph.

We introduce one final generalisation and then provide the algorithm. Since our goal may sometimes be only to provide a bound on the final iteration in an interval of iterations in a particular scenario, supposing we know how long the interval lasts, we may want to suppress enforcing the upper bound inside the interval or beyond the end of that interval. In order to be able to do this, we provide an additional parameter m to the algorithm, indicating that we only want to enforce the constraint starting from iteration m. In particular m is often equal to 1, because we know that we will execute at least one iteration. The algorithm to compute the bounds is given in Algorithm 4.

The algorithm is initially called as follows: BOUNDS($\mathbf{G}, \overline{\gamma}^*, m, \overline{\gamma}_0$). $\mathbf{G}$ is the

Algorithm 4 Compute delay-period trade-off

1: BOUNDS($\mathbf{G}, \overline{\gamma}^*, m, \overline{\gamma}, k = 0, S = \varnothing$)
2: **if** there is $(\overline{\gamma}_s, l) \in S$ with ${\overline{\gamma}_s}^{norm} = \overline{\gamma}^{norm}$ **then**
3: **return** $\{\pi \geq ||\overline{\gamma} - \overline{\gamma}_s||/(k-l)\}$
4: **else**
5: $S \leftarrow S \cup \{(\overline{\gamma}, k)\}$
6: $\overline{\gamma}' \leftarrow \mathbf{G}\overline{\gamma}$
7: **if** $k < m$ **then**
8: **return** BOUNDS($\mathbf{G}, \overline{\gamma}^*, m, \overline{\gamma}', k+1, S$)
9: **else**
10: $DPE \leftarrow \{\tau_i + k \cdot \pi \geq \overline{\gamma}(i) - \overline{\gamma}^*(i) \mid 1 \leq i \leq size(\overline{\gamma})\}$
11: **return** $DPE \cup$ BOUNDS($\mathbf{G}, \overline{\gamma}^*, m, \overline{\gamma}', k+1, S$)
12: **end if**
13: **end if**

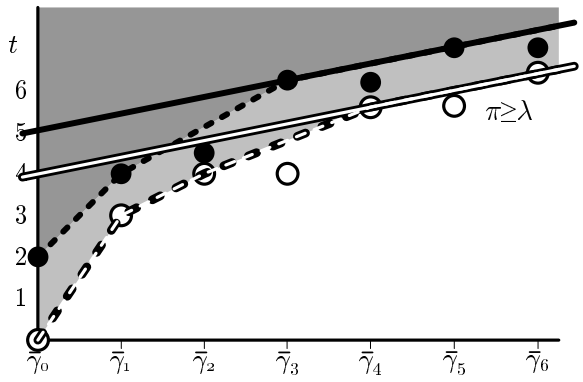

Fig. 7. Linear upper bound equations

SDFG. $\overline{\gamma}^*$ is the reference schedule. $\overline{\gamma}_0$ is the initial condition from which the graph starts. From iteration m, the bound is enforced. The algorithm returns a set of inequalities, the solutions of which are the upper bound models $(\overline{\tau}, \pi)$ and the Pareto points are tight ones.

In the algorithm, S is a set of schedules observed so far and is used to detect a recurrent state (they can be efficiently found in S), at which point the analysis can end (line 3) with the final equation on the minimum period (maximum throughput) of the graph. In every step the algorithm stores the current schedule (line 5), computes the next schedule by executing one iteration (line 6), starting from iteration m, enforces the constraints and continues with a recursive call (lines 7-12). The new equations are added in line 10. Note that as many equations are added as there are entries in the vectors. τ_i refers to entry i of the delay vector $\overline{\tau}$. Every equation is a constraint involving one of the entries of $\overline{\tau}$ and the period π. With linear programming techniques, redundant equations can be removed if necessary and the desired solutions can be found.

6. EXECUTION TIME ANALYSIS

In this section we show how the steady-state spectral analysis and the transient analysis of the previous section can be combined to a method for analysing SDF scenarios.

6.1 Proving Bounds on Scenarios

The monotonicity property of SDFG execution enables an assume-guarantee style reasoning for proving upper bounds on the production times of output: if we have an upper bound $\overline{\gamma}$ on the times when the initial tokens for an iteration become available, then $\mathbf{G}\overline{\gamma}$ is guaranteed to be an upper bound on the times when the new tokens are made available after that iteration. This bound is in turn used as the starting assumption for the next iteration and so on and so forth. For longer sequences of iterations in the same scenario, we look for a schedule that can be used as an invariant schedule throughout the interval. An eigenvector of the scenario graph is of course the obvious candidate for such an invariant. In conjunction with the results obtained from Algorithm 4, this gives us a powerful and flexible tool to establish bounds on worst-case behaviour of SDF scenarios. We illustrate this by several case studies in Section 7.

Besides the analysis of intervals of iterations in a particular scenario, we also need to analyse the transitions from one scenario interval to another. We can do this analysis for any specific combination of scenarios. Analysis of the first scenario interval provides us with a bound on the completion of the last iteration in that scenario. We can use Algorithm 4 to analyse how the second scenario interval behaves when started from this bound.

A disadvantage of the approach explained above, can be that any combination of two scenarios has to be analysed individually. In case there are many scenarios, this may be a problem. A more efficient approach to analysis of SDF scenarios is to establish a single reference schedule, $\overline{\gamma}^*$, which is used as an invariant in-between scenario intervals. Then, for each scenario $s \in \Sigma$, we can establish a duration τ_s such that if we assume all tokens are available at $\overline{\gamma}^*$, we can guarantee that the new tokens are produced no later than $\tau_s + \overline{\gamma}^*$. In the same way, if we have a desired period π_s for this scenario, we can determine a tight τ_s such that the results of k iterations are produced no later than $\tau_s + k \cdot \pi_s + \overline{\gamma}^*$ This way, we obtain a simple model for estimating the performance of these SDF scenarios, namely we can simply add τ_s for each interval executed in scenario s and the number of iterations in that interval times the period. The experiments in the next section show that this approach is often sufficient.

6.2 Optimal Reference Schedule

The approach introduced above raises the question of how to determine the invariant schedule. The smaller the entries in the reference vector are chosen, the more difficult it becomes to establish it at the end of an interval. A higher value on the other hand makes the starting condition of the interval less favourable. Both could lead to extra delay. Which schedule is optimal may even depend on the frequency of occurrence of specific scenarios, by favouring scenario transitions that occur frequently. Although determining the optimal reference schedule is a difficult problem

in general, which requires further study, the case studies in the next section show that a good schedule is often relatively easy to find in practical cases. We have used the following heuristic. Firstly, rather than normalising all eigenvectors to have a norm of 0, we align them on the critical tokens (if the same tokens are critical in different scenarios). We use alap schedules of scenarios, because the late production times potentially allow for shorter initial delay, while at the same time, being alap, they are still soon enough to make the same deadlines in the long run, the eigenvector of the scenario. Since we want to guarantee that we make the alap deadline, regardless of the next scenario, we use as a reference schedule:

$$\overline{\gamma}^* = \min_{s \in \Sigma} \overline{\gamma}_s^L$$

In this selection of reference schedules no additional penalty in scenario switches are encountered in the case studies.

We can generalise the notion of an eigenvector of a single scenario to a set of scenarios as follows. Let $\overline{\gamma}^{S*}$ be the smallest fixed-point of the equation:

$$\overline{\gamma}^{S*} = \max_{s \in \Sigma} \mathbf{G}_s \overline{\gamma}^{S*} - \lambda_s$$

(Note that for a single scenario, this equation reduces to the asap eigenvalue equation.) This fixed-point can be shown to exist, because of the monotonicity of the Max-Plus operations, if we include ∞ as a least upper bound of $\mathbb{R}$. It is unique up to addition of a constant (uniform shifts in time).

PROPOSITION 6.1. *If* $||\overline{\gamma}^{S*}|| < \infty$*, then for every scenario* $s \in \Sigma$*,* $(\overline{\gamma}^{S*}, \lambda_s)$ *is a delay-period model of execution of* $\mathbf{G}_s$ *starting from* $\overline{\gamma}^{S*}$

PROOF. If follows directly from the definition of $\overline{\gamma}^{S*}$ that $\mathbf{G}_s \overline{\gamma}^{S*} \preceq \overline{\gamma}^{S*} + \lambda_s$ and thus by repeating the argument $\mathbf{G}_s^k \overline{\gamma}^{S*} \preceq \overline{\gamma}^{S*} + k\lambda_s$. □

It is a sufficient condition, but not necessary. It follows from the proposition that if the fixed-point exists, then with this schedule as a reference schedule, there are no (positive) delays involved with the transition from one scenario to another. It is only a useful reference schedule if it has a norm smaller than ∞. A similar result can be proved for the alap fixed-point schedule. For the case studies in the next section, $\overline{\gamma}^{S*}$ and the alap fixed-point coincide.

7. CASE STUDIES

The Max-Plus based exploration of Algorithm 2, Algorithm 4 computing the trade-offs in SDF transient behaviour and the spectral analysis in Algorithm 3 have been implemented as an extension of the publicly available SDF^3 tool set [Stuijk et al. 2006] and have been used in the case studies presented in this section. It is known that although state-space exploration can take a long time in the worst-case [Ghamarian et al. 2006; Stuijk et al. 2006], it is mostly well-behaved in practical cases. Run-times for the analyses used for the case studies in this paper are all well below one second.

7.1 MPEG-4 AVC Object Shape Decoder

We first consider a graph in which only execution times change per scenario and try to establish an estimate of the completion time of a given number of iterations

Table I. Execution times MPEG-4 AVC

scenario	s_1	s_2	s_3
η_s	1.858 ms	4.788 ms	6.055 ms
φ_s	0.980 ms	0.980 ms	0.980 ms
β_s	3.816 ms	3.816 ms	3.816 ms

Table II. Steady states of the MPEG-4 AVC decoder

scenario	s_1	s_2	s_3
eigenvalue	3.816 ms	5.768 ms	7.035 ms
eigenvector	-10.468 ms	-7.632 ms	-7.632 ms
	-7.632 ms	-9.584 ms	-10.851 ms
	-3.816 ms	-3.816 ms	-3.816 ms
	-3.816 ms	-3.816 ms	-3.816 ms
	-3.816 ms	-5.768 ms	-7.035 ms
	0.000 ms	0.000 ms	0.000 ms
	0.000 ms	0.000 ms	0.000 ms
τ_s	0.000 ms	0.000 ms	0.000 ms
τ_s^0	6.654 ms	7.632 ms	7.632 ms

using information about the number of iterations in various scenarios. We illustrate this by redoing the analysis of [Poplavko et al. 2007] with our techniques. The graph of Figure 2 represents the behaviour of the performance-critical part of an MPEG-4 AVC decoder mapped onto an SoC platform (see [Poplavko et al. 2007] for the details). The graph captures the processing of three different types of image blocks together with the pre-fetching of the required data from a memory across a network-on-chip. An HSDF model is derived in [Poplavko et al. 2007] and subsequently simplified and shown to be equivalent to the graph of Figure 2. The actor execution times φ_s and β_s are constant (independent of the scenarios) and η_s varies for the three scenarios s_1, s_2 and s_3, as shown in Table I.

It is shown in [Poplavko et al. 2007] how scenario information embedded in the video stream meta data can enable a tight estimation of the frame decoding completion time. Scenarios are based on types of blocks. The number of iterations (individual block decoding) is provided as well as information about the number of transitions from a particular scenario to another. Hence, implicitly also the number of intervals in which a particular scenario remains active. [Poplavko et al. 2007] shows how a bound on the frame completion time can be given as a linear combination of these parameters.

Table II shows the steady state behaviour of the individual scenarios. The eigenvalues show at what rate iterations can be performed in each scenario. The eigenvectors show the relative production times of the tokens in the steady state within one iteration. The vectors are normalised, so that the maximum entry is 0. The order is according to the numbering of the initial tokens in Figure 2. Note that the range of time stamps spans more than the repetition period of the iterations, indicating the pipelined execution of the iterations.

In the example, a frame consists of 60 macro blocks, 60 iterations of the graph in different scenarios. We use $\overline{\gamma}_0$ to denote the initial state which is assumed to be the zero vector $\overline{\gamma}_0 = \overline{\mathbf{0}}$, i.e., all initial tokens are assumed to be present at time 0

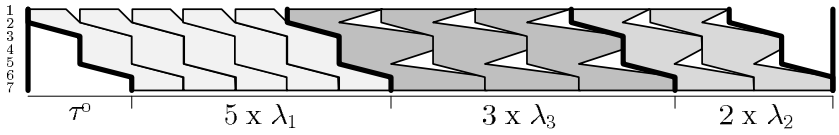

Fig. 8. Bound on scenario intervals

(all buffers empty, resources free). The completion time of the frame is given by the highest entry in the last vector $\overline{\gamma}_{60}$, this is $||\overline{\gamma}_{60}||$.

The execution can be viewed as a sequence of intervals during which only blocks of a specific scenario are processed.

We have to set a target period and a reference schedule for each scenario to apply the approach of Section 6.1. Because we want to establish a tight bound on the frame completion time and we don't know the length of the intervals, we choose the shortest period which can be sustained during an interval of arbitrary length, namely the eigenvalue of the corresponding scenario. We further choose a reference schedule which serves as an invariant in-between iterations. We pick the alap schedule of the execution of the graph. The alap schedule, in this case, is identical for all three scenarios and equals:

$$\begin{aligned}\overline{\gamma}^* \;=\; [&-7.632\text{ms};\;\; -7.632\text{ms};\;\; -3.816\text{ms};\;\; -3.816\text{ms};\\ &-3.816\text{ms};\; 0.000\text{ms};\; 0.000\text{ms}]^T\end{aligned}$$

The fact that this vector is identical for each scenario simplifies the analysis, because we do not need to investigate specific scenario transitions separately and count the number of transitions from one scenario to another.

Let all iterations m in interval $[k\;\; k+N)$ be executed in scenario s. Assume that $\overline{\gamma}_k \preceq T_k + \overline{\gamma}^*$ at the beginning of the interval. We try to find a bound on the end of the interval, i.e. T_{k+N} such that $\overline{\gamma}_{k+N} \preceq T_{k+N} + \overline{\gamma}^*$.

BOUNDS$(\mathbf{G}_s, \overline{\gamma}^*, 1, \overline{\gamma}^*)$ gives us a value of τ_s such that during and after the interval $(k \leq m \leq k+N)$

$$\overline{\gamma}_m \preceq T_k + \tau_s + (m-k)\cdot\lambda_s + \overline{\gamma}^*$$

For the end of the interval we can thus derive that

$$\overline{\gamma}_{k+N} \preceq T_k + \tau_s + N\cdot\lambda_s + \overline{\gamma}^*$$

In other words, if the start of the interval is characterised by $T_k + \overline{\gamma}^*$, then the end of the interval is characterised by $T_{k+N} + \overline{\gamma}^* = T_k + \tau_s + N\cdot\lambda_s + \overline{\gamma}^*$.

Thus, for every interval we have an initial penalty of τ_s and further the length N of the interval times the eigenvalue λ_s of the graph of scenario s. The eigenvalues and the values of τ_s are shown in Table II. Note that the initial 'penalties' are in fact zero for this graph, though they need not be in general and they may even be negative.

The first iteration does not start from $\overline{\gamma}^*$, but starts from the initial state $\overline{\gamma}_0 = \overline{\mathbf{0}}$. With Algorithm 4, we determine the initial delay τ_s^0 for scenario s, also shown in Table II. In case it is not known which is the initial scenario, we can take the maximum of the three.

Table III. Eigenvalues (maximum cycle means) of the MPEG-4 SP application in various scenarios

scenario	**I**	**P99**	**P80**	**P70**	**P60**	**P50**	**P40**	**P30**
eigenvalue	3960	3960	3200	2800	2400	2000	1600	1200

The overall estimate of execution time is as follows. We add up: (i) the number of iterations in a scenario times the eigenvalue of that scenario, (ii) the number of intervals times the transition delay (0 ms for all scenarios in this example), (iii) the initial delay for the first iteration. Note that because the vectors are normalised (the maximum value is zero), there is no need to account for the production times in the final vector. As a result, the following formula computes a guaranteed completion time (with J_s the total number of iterations executed in scenario s and K_s the number of intervals of scenario s).

$$\begin{aligned} T &= \textstyle\sum_{s\in\Sigma} J_s\lambda_s + \sum_{s\in\Sigma} K_s\tau_s + \max_{s\in\Sigma}\tau_s^0 \\ &= 12\cdot 3.816 + 46\cdot 5.768 + 2\cdot 7.035 + 0.000 + 7.632 \\ &= 332.8\text{ms} \end{aligned}$$

The approach is illustrated in Figure 8. For simplicity it only shows three intervals with in total 10 iterations. First 5 iterations in scenario s_1, then 3 in s_3 and finally 2 in s_2. Time is depicted horizontally and the seven tokens in the vectors vertically. The entries of the vectors are connected by lines to visualise them. The bold lines in-between the intervals represent the alap schedule used as a reference. The individual shapes per iteration visualise the following. Their left edge is the smallest alap schedule that fits to the previous schedule and the right edge represent the schedule according to which the new tokens are produced after one iteration. As mentioned above, in this application, iterations repeat at a rate according to the eigenvalue of the scenario and there is no additional delay at the beginning of the interval.

In [Poplavko et al. 2007] the maximum execution times of the actors are re-estimated for every frame and the frame completion is estimated based on these data. For this the relevant behaviour of the graph is computed as a function of these execution times. Because existing tools for analysing SDFGs parametrically are limited [Ghamarian et al. 2008], this is done by hand in [Poplavko et al. 2007]. A similar approach is possible in our framework, but also then proofs need to be given by hand instead of tools. The method in [Poplavko et al. 2007] is only presented for homogeneous data-flow graphs and between scenarios, only the execution times of actors can vary. In the following examples we show that with our technique we can also analyse scenarios of arbitrary SDFGs.

7.2 MPEG-4 SP Decoder

The graph of Figure 1 shows the scenario graphs of an MPEG-4 Simple Profile decoder. In the I-frame scenario, x equals 99 and the motion compensation task (MC) is not used. The P-frame scenario is further split into sub scenarios depending on the number of macro blocks to decode.

Table III shows the eigenvalues of the graph for each of the scenarios and Figure 9 shows the asap and alap eigenvectors. They are depicted as lines which show horizontally the entries of the vector corresponding to the 9 initial tokens and

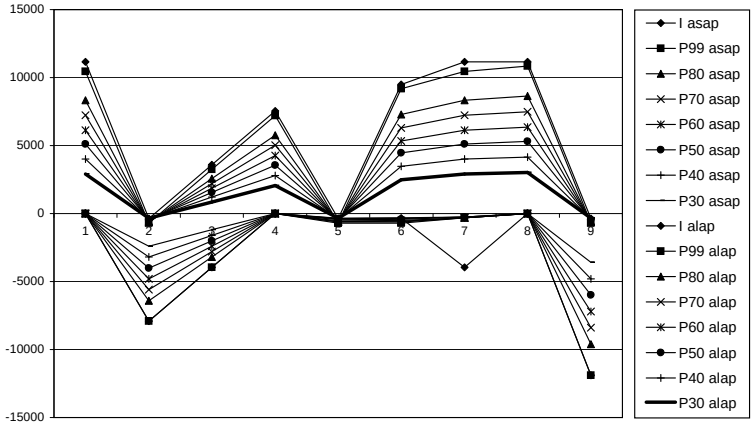

Fig. 9. Scheduling MPEG-4 decoder

vertically the values of the entries. The alap vectors are normalised to the smallest vector above the asap eigenvector. Figure 9 shows that token 5 is critical as it has the smallest distance between alap consumption and asap production schedules. It indicates that the VLD task in the graph (token 5 is a token on a self edge of this actor) is the critical factor. It shows that in particular on the longer scenarios (with a lot of macro blocks), this gives larger amounts of slack to the other tasks. The bold line is the alap reference schedule as explained in Section 6, which coincides with the alap schedule for the P30 scenario.

We have verified using Algorithm 4 that with this alap reference schedule $\overline{\boldsymbol{\gamma}}^*$, all transition delays are 0, $\overline{\boldsymbol{\gamma}}_k \preceq 0 + \overline{\boldsymbol{\gamma}}^* + k\lambda_s$ for every scenario s and $k \geq 0$. With this result and with additional information about the number of occurrences of the scenarios (known or estimated) a bound or estimate on the completion time for decoding a number of frames can be computed. Alternatively, it follows straightforwardly from this analysis that a strictly periodic decoding of 1 frame for every 3960 time units (the largest of the eigenvalues among the scenarios) is guaranteed to be achieved, because for every scenario s

$$\overline{\boldsymbol{\gamma}} \preceq \overline{\boldsymbol{\gamma}}^* \;\Rightarrow\; \mathbf{G}_s\overline{\boldsymbol{\gamma}} \preceq \overline{\boldsymbol{\gamma}}^* + 3960$$

and thus if $\overline{\boldsymbol{\gamma}}_0 \preceq \overline{\boldsymbol{\gamma}}^*$, then for every $k \geq 0$, $\overline{\boldsymbol{\gamma}}_k \preceq \overline{\boldsymbol{\gamma}}^* + k \cdot 3960$.

This example shows that the method can deal with changing consumption / production rates and number of firings of an actor per iteration, including 0, i.e. not being used at all in that iteration. The next case-study even has entirely different structures of the SDF graph in different scenarios.

7.3 MP3 Decoder

Figure 10 shows the structure of an MPEG-I Layer III (MP3) decoder [Shlien 1994]. After decoding the entropy coded bitstream in the Huffman decoder, encoded audio data is obtained in frames of 26ms of audio. Each frame is again separated in individual 'granules'. There are different ways a granule can be encoded. For each channel, left or right, the data can either be split in a large number (96) of small sub bands (short blocks) or a smaller number (32) of larger sub bands (long blocks).

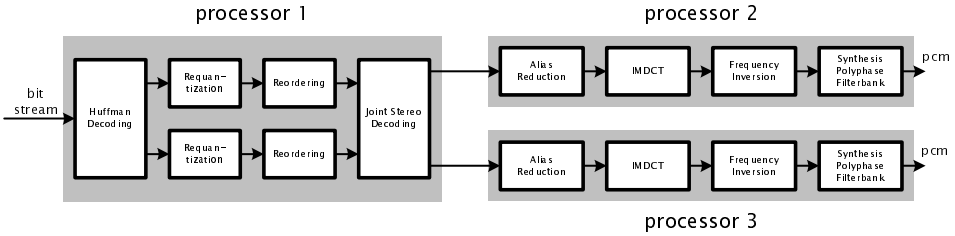

Fig. 10. Structure of an MP3 decoder

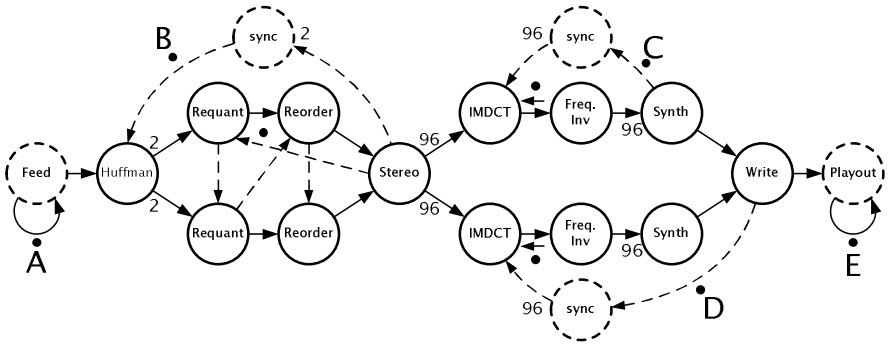

Fig. 11. Mapped MP3 decoder, short scenario

A mixed mode is also possible, in which the spectrum is divided into 2 long and 90 short blocks. In mixed mode, both channels, left and right are encoded in mixed mode. For the other modes the channels can each have their own mode. This gives a set of five scenarios in total, denoted as $\Sigma = \{\text{s-s}, \text{l-l}, \text{l-s}, \text{s-l}, \text{m}\}$. In [Gheorghita et al. 2006], it is shown that the encoding modes are good candidates for scenarios of the decoding process on a single processor system. Here we study an (hypothetical) multiprocessor implementation. We assume that the Huffman decoding up to the Joint Stereo Decoding tasks are performed on one processor, the left channel tasks on a second processor and the right channel tasks on a third. One of the processors also takes care of writing back the decoded audio into a playout buffer. Which one depends on the scenario. The Reorder tasks are only executed for short blocks, while the Alias Reduction tasks are only executed on long blocks. This is the main reason why an SDF model of the application, not exploiting scenarios leads to a too pessimistic assessment of the worst-case performance of the application.

We have made separate SDF models of the decoding modes, shown in Figures 11 (s-s), 12 (l-l) and 13 (m) for short blocks, long blocks, and mixed mode respectively. There are two more scenarios (s-l) and (s-l) which are not shown separately, but combine decoding long and short blocks for different channels, similar to the graphs in Figures 11 and 12.

The SDF models capture the MP3 decoding tasks, but also the input bit stream and the playout of decoded audio with additional Feed and Playout actors. One iteration of each of the graphs corresponds to the decoding of one frame of audio.

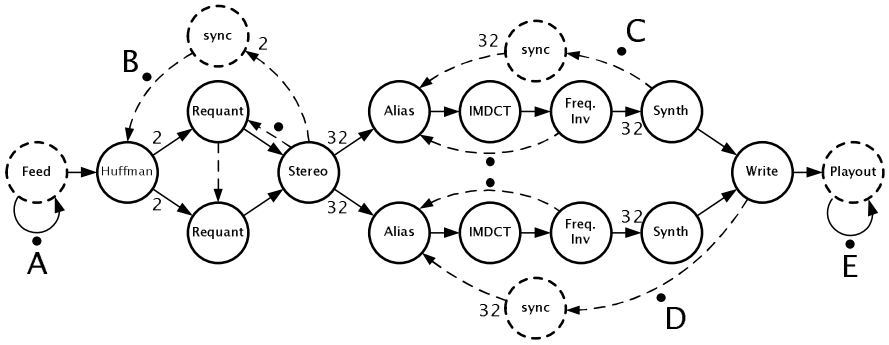

Fig. 12. MP3 decoder, long block scenario

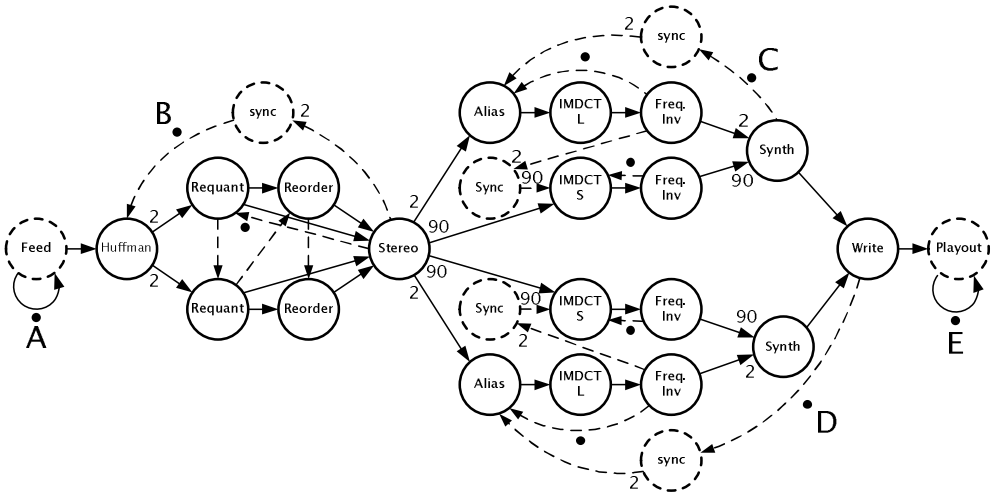

Fig. 13. MP3 decoder, mixed block scenario

Note that there are no data-dependencies between subsequent frames. The only dependencies between frames in this model arise from the fact that they are decoded by the same processor. The tokens in the models that carry dependencies from one iteration to the next are labelled ‘A’ up to ‘E’ in each of the models. Tokens ‘B’, ‘C’ and ‘D’ each represent availability of one of the three processors. ‘A’ and ‘E’ model the periodic feed of data to the decoder and playout of decoded audio. Besides the actors modelling the decoding tasks, some actors have been added to capture the processor mapping and scheduling. Dashed dependencies and Sync actors, with execution time 0, are added to model the processor schedule. They enforce a particular order in which actors are executed sequentially on their processor. For example, in the mixed-mode scenario of Figure 13, in the left channel, one can check that the schedule consists of two times the sequence Alias—IMDCT L—Freq.Inv. followed by 90 iterations of the sequence IMDCT S—Freq.Inv. followed by one execution of Synth. This whole sequence is executed twice (for two

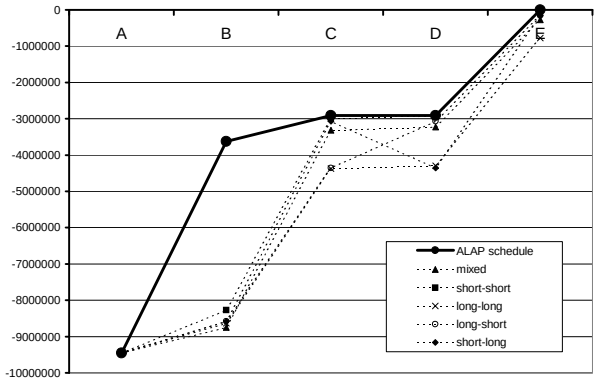

Fig. 14. Scheduling MP3 processors

granules). This last firing completes the iteration and returns the token 'C'. This carries over the processor dependency to the next iteration, i.e., the processor can only start on the next frame after completing the current. The other tokens do not carry dependencies between iterations and are only used to express the sequence constraints of the processor mapping. For the transient analysis they can be set to $-\infty$ so as to not constrain the firing times.

The MP3 decoder has been profiled [Gheorghita et al. 2006] on an ARM7 [ARM] processor and we use these worst-case cycle requirements for each of the actors on their respective processors. With this information, we can perform spectral analysis on each of the individual scenarios, giving us the minimum number of processor cycles per period in an execution in that scenario (the eigenvalue) as well as the corresponding schedule (the eigenvector). The results show that the slowest of the scenarios are the scenarios involving short block s-s, s-l and l-s, with an upper bound of 5.829 MCycles per iteration. For the sake of this example, we assume that we configure the processors to run at a speed of 5.829 MCycles per iteration, which corresponds to 26 ms frames.

Although we know that each of the individual scenarios can run at this speed if all iterations are of that same scenario, we still need to demonstrate that this is also sufficient when facing arbitrary orders of scenarios. To do this, we need a suitable reference schedule and prove using Algorithm 4 that the model is able to switch from one scenario to the next without additional switching delays. We use again the alap firing times in maximum throughput executions of each of the scenario SDFs and call it $\overline{\gamma}^*$, it is the bold line in Figure 14.

Different from the previous case studies we now use delay-period models, not necessarily with the minimal period of the eigenvalue, but instead with a fixed period of the 5.829 MCycles which are available per frame. With this reference schedule as an initial condition and this given period, we have analysed the transition penalties, i.e. the initial delay with $\overline{\gamma}^*$ as a starting condition as well as the reference schedule. The results show that for each of the scenarios, this penalty is in fact 0. The same analysis with as initial vector the zero vector $\overline{\mathbf{0}}$ shows, for each scenario, an initial delay of $\tau^0 = 9.456$ MCycles, so that it is proved that no matter what sequence of

scenarios are encountered,

$$\overline{\gamma}_k \preceq \tau^0 + k \cdot \pi + \overline{\gamma}^*$$

This proves that the MP3 decoder can indeed meet all deadlines and produces a new frame every 26 ms, whatever sequence of scenarios may occur. As a comparison we have made a single SDF model that represents the worst-case over all scenarios, but it has a minimum period of 9.28 MCycles, which shows that the scenario based analysis allows for proving tighter worst-case execution bounds. Based on the static SDF model one would be forced to run the processor a a higher speed to be sure to meet all real-time constraints.

Figure 14 shows that particularly token B which corresponds to processor 1 has quite a bit of slack (distance between asap and alap schedules). The analysis provides more useful information. The alap schedule provides deadlines for the individual processors to guarantee that the graph meets the final deadline of the playout. This deadline per processor can be used for instance by a scheduler to schedule other tasks of other applications on the processor, or to use voltage and frequency scaling techniques to reduce power consumption. As long as the scheduler guarantees that this processor completes its own tasks according to this schedule, the system as a whole will produce all output in time.

8. DISCUSSION AND CONCLUSION

We have introduced novel techniques for worst-case performance analysis of data-flow applications on embedded multiprocessor platforms. The approach builds upon model-based design and scenario-based analysis concepts and captures such systems by scenarios, each characterised by its own Synchronous Data-Flow graph. It is shown how combining analysis of steady-state behaviour of scenarios and transient behaviour of scenario changes can provide guarantees on the worst-case performance.

The worst-case performance guarantees can be further exploited in various ways. They allow for optimising the platform, i.e. finding the minimum amount of resources to satisfy performance requirements. They can be integrated into design-space exploration and application mapping techniques. They can be used for run-time resource management and optimisation, using the fact that SDFGs can often conveniently model resource allocation [Sriram and Bhattacharyya 2000].

If there are reconfigurations which cannot take place in a pipelined fashion, such as global reconfiguration or resource reallocation, then techniques from this paper can be used to assess the time it takes to complete all ongoing iterations and after reconfiguration restart the new ones.

In future work we intend to explore the opportunities for exploiting the analysis results. We want to extend the (automated) scenario classification methods and on-line scenario exploitation methods developed for sequential software [Gheorghita et al. 2006] to a multiprocessing context. We further want to explore the possibilities of more accurate analysis techniques in between the monolithic approach of SADF and the more conservative compositional approach of this paper.

ACKNOWLEDGMENT

I would like to thank Bart Theelen en Peter Poplavko for valuable discussions on the subject of this paper, as well as the anonymous reviewers for their constructive feedback.

REFERENCES

ARM. Arm7tdmi processor. `http://www.arm.com/products/CPUs/ARM7TDMI.html`.

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

Bhattacharya, B. and Bhattacharyya, S. 2001. Parameterized dataflow modeling for DSP systems. *IEEE Transactions on Signal Processing 49,* 10 (October), 2408–2421.

Bilsen, G., Engels, M., Lauwereins, R., and Peperstraete, J. 1996. Cyclo-static dataflow. *IEEE Transactions on signal processing 44,* 2 (February), 397408.

Boudec, J. L. and Thiran, P. 2003. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Lecture Notes in Computer Science, vol. 2050. Springer.

Buck, J. 1993. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. Ph.D. thesis, University of California, EECS Dept., Berkeley, CA.

Chao, L.-F. and Sha, E. H.-M. 1995. Static scheduling for synthesis of dsp algorithms on various models. *J. VLSI Signal Process. Syst. 10,* 3, 207–223.

Cochet-Terransson, J., Gaubert, S., and Gunawardena, J. 1999. A constructive fixed point theorem for min-max functions. *Dynamics and Stability of Systems 14,* 4 (June), 407–433.

Dasdan, A., Irani, S. S., and Gupta, R. K. 1999. Efficient algorithms for optimum cycle mean and optimum cost to time ratio problems. In *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*. ACM, New York, NY, USA, 37–42.

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

Ghamarian, A., Geilen, M., Basten, T., and Stuijk, S. 2008. Parametric throughput analysis of synchronous data flow graphs. In *Proceedings of the Design Automation and Test in Europe Conference (DATE)*. Munich, Germany, 116–121.

Ghamarian, A., Geilen, M., Stuijk, S., Basten, T., Moonen, A., Bekooij, M., Theelen, B., and Mousavi, M. 2006. Throughput analysis of synchronous data flow graphs. In *Proceedings of the 6th International Conference on Application of Concurrency to System Design (ACSD'06)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 2006, 27–30.

Ghamarian, A. H., Geilen, M. C. W., Basten, T., Theelen, B. D., Mousavi, M. R., and Stuijk, S. 2006. Liveness and boundedness of synchronous data flow graphs. In *FMCAD '06: Proceedings of the Formal Methods in Computer Aided Design*. IEEE Computer Society, Washington, DC, USA, 68–75.

Gheorghita, S., Basten, T., and Corporaal, H. 2006. Application scenarios in streaming-oriented embedded system design. In *International Symposium on System-on-Chip 2006, SoC 2006, Proc.* IEEE, 175–178.

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

Heidergott, B., Olsder, G. J., and van der Woude, J. 2006. *Max Plus at Work*. Princeton University Press.

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

Karp, R. M. and Miller, R. E. 1966. Properties of a model for parallel computations: Determinancy, termination, queueing. *SIAM Jounal of Applied Mathematics 14,* 6 (Nov.), 1390–1411.

Künzli, S., Hamann, A., Ernst, R., and Thiele, L. 2007. Combined approach to system level performance analysis of embedded systems. In *CODES+ISSS '07: Proceedings of the 5th*

IEEE/ACM international conference on Hardware/software codesign and system synthesis. ACM, New York, NY, USA, 63–68.

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

Lee, E. and Sangiovanni-Vincentelli, A. Dec 1998. A framework for comparing models of computation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 17,* 12, 1217–1229.

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

Moreira, O. M. and Bekooij, M. J. G. 2007. Self-timed scheduling analysis for real-time applications. *EURASIP Journal on Advances in Signal Processing*.

Poplavko, P., Basten, T., and van Meerbergen, J. 2007. 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*. IEEE Computer Society, Washington, DC, USA, 228–235.

Sangiovanni-Vincentelli, A. and Martin, G. 2001. Platform-based design and software design methodology for embedded systems. *IEEE Des. Test 18,* 6, 23–33.

Shlien, S. Dec 1994. Guide to mpeg-1 audio standard. *Broadcasting, IEEE Transactions on 40,* 4, 206–218.

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

Stefanov, T., Zissulescu, C., Turjan, A., Kienhuis, B., and Deprettere, E. 2004. System design using kahn process networks: The compaan/laura approach. *date 01*, 10340.

Stiliadis, D. and Varma, A. 1998. Latency-rate servers: a general model for analysis of traffic scheduling algorithms. *IEEE/ACM Trans. Netw. 6,* 5, 611–624.

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

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

Thiele, L., Chakraborty, S., and Naedele, M. 2000. Real-time calculus for scheduling hard real-time systems. In *International Symposium on Circuits and Systems ISCAS 2000*. Vol. 4. Geneva, Switzerland, 101–104.

Wiggers, M., Bekooij, M., and Smit, G. J. M. 2007. Efficient computation of buffer capacities for cyclo-static dataflow graphs. In *DAC*. 658–663.

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