Parametric Throughput Analysis of Scenario-Aware Dataflow Graphs

Morteza Damavandpeyma[1], Sander Stuijk[1], Marc Geilen[1], Twan Basten[1,2] and Henk Corporaal[1]
[1]Department of Electrical Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands
[2]Embedded Systems Institute, Eindhoven, The Netherlands
{m.damavandpeyma, s.stuijk, a.a.basten, m.c.w.geilen, h.corporaal}@tue.nl

***Abstract*—Scenario-aware dataflow graphs (SADFs) efficiently model dynamic applications. The throughput of an application is an important metric to determine the performance of the system. For example, the number of frames per second output by a video decoder should always stay above a threshold that determines the quality of the system. During design-space exploration (DSE) or run-time management (RTM), numerous throughput calculations have to be performed. Throughput calculations have to be performed as fast as possible. For synchronous dataflow graphs (SDFs), a technique exists that extracts throughput expressions from a parameterized SDF in which the execution time of the tasks (actors) is a function of some parameters. Evaluation of these expressions can be done in a negligible amount of time and provides the throughput for a specific set of parameter values. This technique is not applicable to SADFs. In this paper, we present a technique, based on Max-Plus automata, that finds throughput expressions for a parameterized SADF. Experimental evaluation shows that our technique can be applied to realistic applications. These results also show that our technique is better scalable and faster compared to the available parametric throughput analysis technique for SDFs.**

I. Introduction

Signal processing and multimedia applications can be modeled with synchronous dataflow graphs (SDFs) [1]–[4]. An SDF can be analyzed to determine performance properties (e.g., throughput [5]) or resource requirements (e.g., buffer sizes [6]) of the underlying application. However, SDFs cannot efficiently capture the dynamic behavior of modern streaming applications (e.g., audio or video codecs with advanced compression schemes) because of their static nature. Using SDFs to model such applications with high dynamism cannot assure tight performance guarantees. Scenario-aware dataflow graphs (SADFs) [7] are have been introduced to relax this limitation of SDFs. An SADF of an application is composed of several SDFs and a finite state machine (FSM). Each mode (scenario) of the application in the SADF is modeled by an SDF; the FSM captures the order of scenario occurrence. In [8], it is shown that SDF throughput analysis (e.g., [5]) can result in pessimistic performance bounds. The paper introduces a novel technique to determine a tighter throughput bound for applications modeled with an SADF. The approach extracts a Max-Plus automaton graph (MPAG) from an SADF and then uses a maximum cycle mean algorithm to determine the critical timing cycle of the extracted MPAG.

The timing behavior of an application depends on its binding, scheduling, buffer allocation, etc. Dynamic voltage and frequency scaling (DVFS), which is a commonly used technique to reduce the energy consumption, has also a direct influence on the timing behavior of an application. Common design space exploration (DSE) frameworks [9]–[11] perform several throughput calculations to determine the performance of multiple solutions in the design space. Moreover, at run-time, throughput calculation might be required when a run-time parameter (e.g., frequency of a processor) is changed. Both at design-time and at run-time, throughput analysis must be performed as fast as possible. To address this challenge, [12] introduces a parametric throughput analysis technique for SDFs. The technique finds throughput expressions for a parameterized SDF in which actors (tasks) can have parameters as their execution time. These parameters have a specified time interval. The combination of all parameters of the SDF forms a multi-dimensional parameter space. A divide and conquer technique is used to determine all throughput regions in the parameter space. Each throughput region corresponds to a critical timing cycle in the SDF. For each region a throughput expression is discovered using a state-space exploration technique. The discovered throughput expressions can be used in any DSE framework or run-time manager to quickly compute the throughput of an SDF when the concrete values for all parameters are known. An evaluation of these throughput expressions (instead of a complete throughput analysis) can be done quickly while providing the same result.

However, the technique from [12] is not directly applicable to SADFs. Applying SDF throughput analysis on applications with a dynamic behavior may result in a loose bound on the worst-case throughput. So, a new technique is required to determine throughput expressions for dynamic applications. This paper presents such a parametric SADF throughput analysis technique. We use several real-world applications to evaluate our technique. Our experiments show that our technique is also better scalable than the one from [12]. The throughput expressions found using our technique can be applied to solve practical problems; we demonstrate how to use our technique to determine the lowest multiprocessor frequency setting under an application throughput constraint.

The remainder of the paper is structured as follows. Sec. II introduces the preliminary concepts. Sec. III explains our proposed technique to determine throughput expressions for parametric SADFs. The proposed technique needs to convert a parameterized SADF into its equivalent symbolic MPAG. The complexity of a MPAG-based analysis technique depends (amongst others) on the number of initial tokens of the model. Some modeling techniques (e.g., buffer size modeling) may add a large amount of initial tokens to the graph. This

This work was supported in part by the Dutch Technology Foundation STW, project NEST 10346.

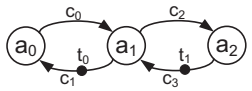


Figure 1. An example SDF.

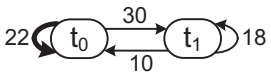
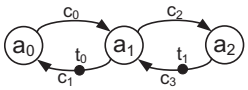


Figure 2. MPAG of the example SDF.



(a) Scenario graph.

(b) FSM.

	a_0	a_1	a_2
scenario A	12	10	8
scenario B	2	18	4

(c) Actor execution times in scenario A and B.

Figure 3. SADF with two scenarios A and B.

can increase the run-time of any technique that relies on MPAGs. In Sec. IV, we introduce a technique to reduce the number of initial tokens in a dataflow graph without changing the behavior of the model. We evaluate our technique on several realistic applications in Sec. V. An application of our technique is presented in Sec. VI. Sec. VII contains the related work. Sec. VIII concludes.

II. Preliminaries

This section introduces the basic concepts and terminology.

A. Synchronous Dataflow Graphs (SDFs)

An SDF is a directed graph (A, C). A node $a \in A$, called *actor*, represents a function (task) of the application. An edge $c \in C$, called *channel*, captures (data) dependencies between actors. Fig. 1 depicts an example SDF with 3 actors ($A = \{a_0, a_1, a_2\}$) and 4 channels ($C = \{c_0, c_1, c_2, c_3\}$). Actors communicate with *tokens* sent from one actor to another over the channels. Channels may contain initial tokens, depicted with a solid dot (and an attached number in case of multiple tokens). The example graph contains two initial tokens which are labeled t_0 and t_1. An essential property of SDFs is that every time an actor *fires* (executes) it consumes the same amount of tokens from its input channels and produces the same amount of tokens on its output channels. These amounts are called the *rates* (indicated next to the channel ends when the rates are larger than 1). The rates in an SDF determine how often actors have to fire with respect to each other such that the distribution of tokens over all channels is in balance. This property is captured in the *repetition vector* [1] of an SDF. Fixed consumption and production rates allow SDFs to execute in a periodic form, which is called an *iteration*. In one iteration each actor is fired as often as indicated in the repetition vector of the SDF. After one iteration of the SDF, the token distribution is guaranteed to return to the initial token distribution. Consistency (i.e., the existence of a repetition vector) and absence of deadlock are practically necessary conditions for SDFs which can be verified efficiently [13], [14]. Any SDF which is not consistent requires unbounded memory to execute or deadlocks. Therefore, we limit ourselves to consistent and deadlock free SDFs.

B. Max-Plus Algebra for SDFs

Let $\tau_a \in \mathbb{N}_0$ be the execution time of an actor $a \in A$. Consider the vector γ_k ($k \in \mathbb{N}$), which is called the *token timestamp* vector. Each entry in γ_k corresponds to the production time of an initial token in the k^{th} iteration of the graph. γ_0 represents the initial token timestamp vector of the SDF. All entries in γ_0 are assumed to be zero. Starting from a vector γ_k and after completing the $(k+1)^{th}$ iteration of the graph, a new vector γ_{k+1} is found. These vectors γ_i can be computed using Max-Plus algebra [15]. For each SDF, a characteristic Max-Plus matrix $G|_{n\times n}$ ($n = |\gamma|$) exists where an entry $G[i, j] \in G$ specifies the minimum time distance from the j^{th} token in the previous iteration to the i^{th} token in the current iteration. Reference [16] explains how to build this matrix for an SDF. Assume now that the actor execution times of the three actors in our example SDF (see Fig. 1) are equal to $\tau_{a_0} = 12$, $\tau_{a_1} = 10$ and $\tau_{a_2} = 8$ time units. The characteristic matrix of our example is then equal to:

$$G = \begin{array}{c} \\ t_0 \\ t_1 \end{array} \begin{array}{c} \begin{array}{cc} t_0 & t_1 \end{array} \\ \begin{pmatrix} 22 & 10 \\ 30 & 18 \end{pmatrix} \end{array}$$

The matrix G shows for the example that the minimum time distance from token t_0 in the k^{th} iteration to token t_1 in the $(k+1)^{th}$ iteration is 30 time units via $a_0 - a_1 - a_2$. When the i^{th} token is not dependent on the j^{th} token, then $G[i, j]$ will be equal to $-\infty$.

Using Max-Plus matrix multiplication, the characteristic matrix can be used to determine the evolution of the token timestamp vector:

$$\gamma_{k+1} = G \otimes \gamma_k \tag{1}$$

As an example, γ_1 can be computed using Eqn. 1 as below:

$$\gamma_1 = \begin{pmatrix} 22 & 10 \\ 30 & 18 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} \max\{22+0, 10+0\} \\ \max\{30+0, 18+0\} \end{pmatrix} = \begin{pmatrix} 22 \\ 30 \end{pmatrix}$$

Using an iterative approach, any timestamp vector γ_k ($k \in \mathbb{N}$) can be calculated. In [8], a technique is proposed to calculate throughput by creating a MPAG from the characteristic matrix. Fig. 2 shows the corresponding MPAG for the example SDF. In a MPAG, a node is created for each initial token in the SDF and if $G[i, j]$ is not equal to $-\infty$ an edge with weight $G[i, j]$ is added from the node of the j^{th} token to the node of the i^{th} token. The critical cycle of the SDF, i.e., the cycle limiting the throughput, can be found by performing a maximum cycle mean (MCM) analysis on the MPAG. The throughput is the inverse of the MCM. In our example SDF, the edge related to $G[0, 0]$ (shown with a bold arrow) determines the throughput which is equal to $1/22$ iterations/time-unit.

C. Scenario-Aware Dataflow Graphs (SADFs)

An SADF models a dynamic application with multiple operating modes (scenarios). Each scenario of the SADF is modeled through an SDF. The SADF model allows different scenarios to use the same or a different SDF. A finite state machine (FSM) is used to specify the order of scenario occurrences. Each state in the FSM corresponds to a scenario of the SADF and each edge in the FSM models a scenario transition. Fig. 3 shows an example SADF with two scenarios A and B. In this example, both scenarios use the same scenario graph, but the execution times of the actors differ in both scenarios (see Fig. 3(c)). When the FSM transitions to an FSM

state, the scenario graph (SDF) associated with the scenario in this state is executed for one iteration. The initial tokens in the scenario graph capture the dependencies between subsequent iterations of the same scenario graph (as is also the case in an SDF) or iterations of different scenario graphs. The timestamp at which these tokens are produced by the k^{th} iteration of the SADF determines the time at which these tokens are available for consumption by the $(k+1)^{th}$ iteration. The relation between the initial tokens in different scenario graphs is established through their label.

As for an SDF, characteristic matrices can be determined for each scenario. The corresponding matrices for scenarios A and B are:

$$G_A = \begin{pmatrix} 22 & 10 \\ 30 & 18 \end{pmatrix} \qquad G_B = \begin{pmatrix} 20 & 18 \\ 24 & 22 \end{pmatrix}$$

Each characteristic matrix can be translated to a MPAG (see Sec. II-B). Reference [8] explains how to combine the MPAGs of all scenarios of an SADF to a single MPAG. Fig. 4 shows the MPAG for our example SADF. Briefly, a node is added to the MPAG for each token in the scenario graph of an FSM state (e.g., node A/t_0 for token t_0 in scenario A in Fig. 4). If $G_M[y,x]$ is not equal to $-\infty$ and there is a state transition from a state in the FSM that executes scenario N to a state that executes scenario M, an edge with weight $G_M[y,x]$ is added from node N/t_x to node M/t_y in the MPAG. Using MCM analysis on this MPAG, the critical timing cycle of the SADF can be determined. In our example, this critical cycle is denoted using the bold arrows ($MCM = (30+18)/2 = 24$). This cycle determines the throughput of the SADF, which for our example is equal to $1/24$ iterations/time-unit. This cycle corresponds to the cycle composed of actor a_1 in scenario B and actors a_0, a_1 and a_2 in scenario A. The throughput analysis technique from [8], which was outlined in this sub-section assumes fixed actor execution times. In the next section, we extend this technique to enable throughput analysis for SADFs in which the execution times of actors are functions of some parameters. Using these parameterized SADFs, our technique can compute a set of expressions that express the throughput of the graph in terms of the parameters.

III. Parametric Throughput Analysis of SADFs

A. Motivation

SADFs efficiently capture the dynamism of applications by using different actor execution times in different scenarios. In any system, actor execution times depend on the platform on which the actors are running. E.g., the frequency of the processors on which the actors are running can alter their execution times. When all actors have a fixed actor execution time, throughput analysis can be performed using the techniques from [5] (in case of SDF) or [8] (in case of SADF). However, when executing a design-time mapping flow, the actor execution times are only fixed near the end of the flow.

Existing mapping flows re-evaluate the throughput of an SADF whenever a mapping decision changes the actor execution times. The impact of this design decision can typically only be assessed after a throughput analysis is performed. Since many design alternatives must be evaluated, it is crucial to have a fast throughput analysis technique. Moreover,

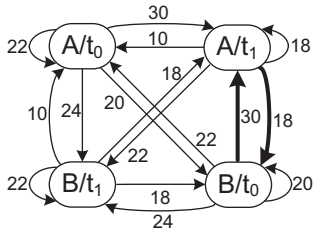


Figure 4. Max-Plus automaton graph of the example SADF.

existing state-of-the-art throughput analysis techniques provide limited information to steer the design decisions (i.e., the critical cycle can be extracted, but it is often not possible to determine how design decisions influence this cycle or any of the other cycles in the graph). To address these issues, [12] introduces a throughput analysis technique for SDFs with parameterized actor execution times. The result of this analysis technique is a set of expressions that can be quickly evaluated once the concrete parameter values are known and that provide the throughput of the SDF for these specific parameter values. In addition, these expressions provide insight in how the parameter values impact the graph's performance. A second advantage of the parameterized throughput analysis technique can be seen when we consider run-time management (RTM). RTM needs to assess whether a new application can be admitted to the system. Typically, it must perform one or more throughput computations. Classical throughput computation usually requires substantial time. For RTM, it is important to take a decision in a short amount of time. The throughput expressions found using the technique from [12] can be used for this purpose as these can be evaluated quickly.

The parametric throughput analysis techniques of [12] are limited to SDFs. We introduce a new technique that can perform parametric throughput analysis for the more expressive SADF model-of-computation. In addition, our technique offers better scalability in terms of the number of parameters that can be used to express the actor execution times.

B. Parametric SADF

A parametric SADF is identical to an SADF except that the actor execution times are not constant. Instead they are a function of a set P of parameters. Each parameter $p_i \in P$ can have any real value within an interval (i.e. $p_i \in I_i = [\min_i, \max_i] \subset \mathbb{R}$). Consider as an example the SADF introduced in Sec. II-C. In a parameterized version of this graph, the actor execution times are a function of two parameters p_1 and p_2. The parameterized execution times of our example parameterized SADF are shown in Tab. I. For simplicity, we assume in our example that the execution time of each actor depends only on a single parameter. Our technique (and implementation) can handle arbitrary linear expressions of the parameters to specify the execution time of each actor, i.e., $c_0 + \Sigma_{i=1}^{|P|} c_i \cdot p_i$ represents the general form of an actor execution time where each $c_i \in \mathbb{R}$ is a constant.

Table I
Actor execution times of the example parametric SADF.

	a_0	a_1	a_2	Parameter range
scenario A	$6p_1$	$2.5p_2$	$2p_2$	$1 \leq p_1 \leq 5$
scenario B	$1p_1$	$4.5p_2$	$1p_2$	$1 \leq p_2 \leq 5$

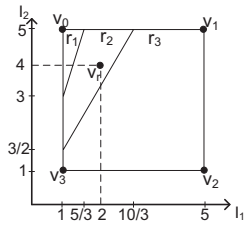


Figure 5. Throughput regions of the example SADF for the parameter space $I_1 \times I_2$. Expressions for the throughput regions: $MCM_1 = 5.5p_2$, $MCM_2 = 3p_1 + 4.5p_2$ and $MCM_3 = 6p_1 + 2.5p_2$.

C. Parameter space and divide & conquer approach

Consider a parameterized SADF that uses a set P of different parameters in the actor execution times. These parameters form a $|P|$-dimensional parameter space which is a convex polyhedron, i.e., $\prod_{i=1}^{|P|} I_i$ where $I_i = [min_i, max_i]$ is the interval to which parameter $p_i \in P$ belongs. Fig. 5 shows a 2-dimensional parameter space for our parametric example SADF. In this figure, the square $\nu_0 - \nu_1 - \nu_2 - \nu_3$ is the initial convex polyhedron. A throughput value $Th(\nu_i)$ can be assigned for each parameter point ν_i inside the parameter space. The throughput of parameter point ν_i can also be calculated by evaluating a throughput expression e_i for the given point ν_i; e.g., $e_r = \frac{1}{3p_1+4.5p_2}$ is the throughput expression for the parameter point $\nu_r : \{p_1 = 2, p_2 = 4\}$ and evaluating the expression e_r for ν_r results in the throughput amount $e_r(\nu_r) = \frac{1}{24}$ for the given parameter point. In [12], it is shown that such a throughput expression e_i can be used to calculate the throughput for all points in a convex sub-polyhedron in the initial parameter space polyhedron (see Proposition 5 from [12]). Hence, the initial parameter space is composed of one or several but a finite number of convex sub-polyhedrons and for each sub-polyhedron a throughput expression exists. Such a sub-polyhedron is called a *throughput region*.

As in [12], we use the same divide & conquer strategy to determine all throughput regions. The parameter space determines the initial polyhedron (e.g., the square in Fig. 5). Initially, a random point ν_r is selected inside the polyhedron. As a first step, the throughput expression e_r for this point must be identified. The algorithm that does this is explained in the next sub-section. Once the throughput expression e_r has been found for this random point ν_r, the divide & conquer technique from [12] evaluates the throughput of each corner point ν_c in the initial polyhedron. When the throughput $Th(\nu_c)$ in a corner point ν_c is equal to the throughput found when evaluating the expression e_r for this point, then this point ν_c belongs to the same throughput region as point ν_r (see Proposition 8 from [12]); if this statement holds for all corner points of a polyhedron, the polyhedron will be identified as a throughput region with throughput expression e_r (see Corollary 6 from [12]). If $Th(\nu_c)$ is not equal to $e_r(\nu_c)$, then it holds that the corner point ν_c belongs to another throughput region than the random point ν_r. In that case, a new throughput expression e_c can be identified for the corner point ν_c. The hyperplane $e_r - e_c = 0$ cuts the initial polyhedron into two convex sub-polyhedrons. For each convex sub-polyhedron the divide & conquer strategy is performed recursively until all throughput regions are identified. The interested reader is referred to [12] for a detailed description of the divide & conquer technique.

D. Throughput expression for a parameter point

In [12], a time-based state-space exploration is used to find the throughput expression for a parameter point ν_r in the parameter space of an SDF. The technique is not directly applicable when more than one scenario in an application exists. Extending a time-based state-space exploration technique for a situation with multiple scenarios is not trivial. In [8], two techniques to compute throughput of an SADF are presented, i.e., an iteration-based state-space exploration and an approach based on MPAG analysis (see Sec. II-C). Inspired by [8], we use an approach based on MPAG analysis to perform parametric throughput analysis of an SADF. The complexity of the approach based on MPAG analysis depends less on the number parameters than the state-space approach. In a MPAG, iteration boundaries across all scenarios are distinguishable; this makes the throughput analysis for SADFs feasible. Algorithm 1 shows the pseudo-code of our algorithm to compute the throughput expression e_r of a parameter point ν_r in a parametric SADF G. In Sec. II-C it is explained how a MPAG can be used to compute the throughput when all actors have a known execution time (i.e., when all parameter values are fixed). We extend this MPAG to a symbolic MPAG that can be used to compute symbolic throughput expressions. As a first step, a symbolic Max-Plus characteristic matrix of each scenario graph must be computed (line 4-5). Next, these matrices must be combined into a symbolic MPAG (line 6). As a third step, the symbolic MPAG is evaluated for a concrete parameter point ν_r. This results in a concrete MPAG (line 7) from which the critical cycle can be extracted using a maximum cycle mean algorithm (line 8). Using a relation between the edges in the concrete and symbolic MPAG, the critical cycle in the concrete MPAG can be translated into a symbolic cycle (expression) in the symbolic MPAG. This symbolic expression is the inverse of the symbolic throughput expression e_r. Algorithm 2 represents our approach to determine the symbolic Max-Plus matrix of a scenario graph, which is an SDF, for a specified parameter point. A symbolic execution of the given graph g - up to one iteration - is performed to determine the token timestamps (lines 1-4 in Algorithm 2). In the symbolic execution, the parameterized actor execution time expressions and a symbolic timestamp for each initial token are used. Fig. 6 illustrates the symbolic execution of scenario B in our example SADF. Step $S.1$

Algorithm 1: Throughput expression for a parameter point

input : SADF G
input : Parameter-Point ν_r
output: Throughput-Expression $\mathrm{e_r}$

1 $\mathrm{n} \leftarrow$ number of scenarios in G
2 $\mathrm{sg}_1 \cdots \mathrm{sg}_n \leftarrow$ scenario graphs of G
3 $\mathrm{fsm} \leftarrow$ FSM of G
4 **for** $i \leftarrow 1$ **to** n **do**
5 $\quad \mathrm{symG}_i \leftarrow$ **getSymG**$(\mathrm{sg}_i, \nu_\mathrm{r})$
6 $\mathrm{symMPAG} \leftarrow$ **getSymMPAG**$(\mathrm{symG}_1 \cdots \mathrm{symG}_n, \mathrm{fsm})$
7 $\mathrm{MPAG} \leftarrow$ **evaluateSymMPAG**$(\mathrm{symMPAG}, \nu_\mathrm{r})$
8 $\mathrm{criticalCycle} \leftarrow$ **maximumCycleMean**(MPAG)
9 **foreach** *channel* $c \in \mathrm{criticalCycle}$ **do**
10 $\quad \mathrm{periodExp} \leftarrow \mathrm{periodExp} +$ **getTerm**$(\mathrm{c}, \mathrm{symMPAG})$
11 $\mathrm{e_r} \leftarrow |\mathrm{criticalCycle}|/\mathrm{periodExp}$

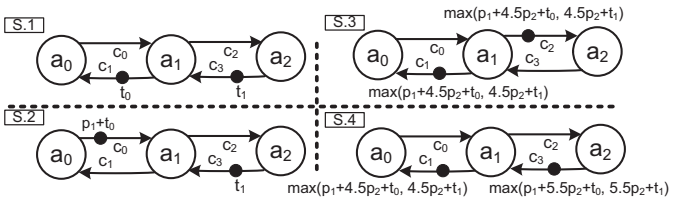


Figure 6. Symbolic execution of scenario graph B of our example SADF.

shows the initial graph and steps $S.2$, $S.3$ and $S.4$ show the graph after consecutive firing of the actors a_0, a_1 and a_2 respectively. Each actor firing is performed symbolically (symbolicFire in Algorithm 2). For example the firing of actor a_0 consumes a token from channel c_1 and produces a token with timestamp $\tau_{a_0} + t_0 = p_1 + t_0$ on channel c_0. Comparing steps $S.1$ and $S.4$ shows that the graph returns to the initial token distribution after one iteration (as expected). The token timestamps in step $S.4$ contain the symbolic dependencies of the initial tokens at the end of this iteration to the initial tokens at the end of the previous iteration. For example, the timestamp $max(p_1+4.5p_2+t_0, 4.5p_2+t_1)$ of the reproduced token t_0 in step $S.4$ implies that token t_0 depends on the production time of the tokens t_0 and t_1 in the prior iteration. The distance of t_0 in the current iteration to t_0 and t_1 in the previous iteration are at least $p_1 + 4.5p_2$ and $4.5p_2$ respectively; this information is used to construct the symbolic Max-Plus characteristic matrix of the scenario graph (line 5 in Algorithm 2). In our example, the matrices for scenarios A and B are equal to:

$$G_A = \begin{pmatrix} 6p_1 + 2.5p_2 & 2.5p_2 \\ 6p_1 + 4.5p_2 & 4.5p_2 \end{pmatrix} \quad G_B = \begin{pmatrix} p_1 + 4.5p_2 & 4.5p_2 \\ p_1 + 5.5p_2 & 5.5p_2 \end{pmatrix}$$

Algorithm 2: Symbolic matrix extraction (**getSymG**)

input : SDF g
input : Parameter-Point ν_r
output: Symbolic Max-Plus Matrix symG

1 **while** *one iteration of g is not completed* **do**
2 **foreach** *enabled actor* $a \in g$ *within one iteration* **do**
3 **symbolicFire**(a)
4 prune timetamps of produced tokens by firing a for the point ν_r
5 extract symG from the resulted timetamps

In practice, it is often not feasible to perform a complete symbolic execution of one iteration of a scenario graph. This is caused by the number of terms that appear in the maximum (max) operator of the token timestamps. In our example, both max operators contain only two terms one for each initial token. The number of terms may however grow rapidly when a graph contains more initial tokens and/or has a large repetition vector rendering a complete symbolic execution impractical. Ref. [17] encounters a similar issue when trying to identify timing expressions for the critical paths in an integrated circuit. This issue is solved by removing redundant expressions which are not affecting the critical paths. In our case, we only need to determine a symbolic Max-Plus characteristic matrix valid for a concrete parameter point (line 7 and 8 in Algorithm 1). This allows us to evaluate the maximum operation (in a token timestamp) for the given parameter point after each symbolic firing of an actor (line 4 in Algorithm 2). In this way, only the terms which have the largest values among all other terms in the maximum operator propagate to the next step of the

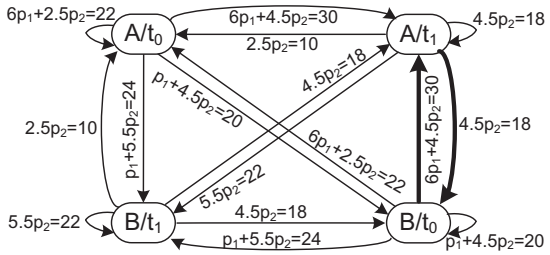


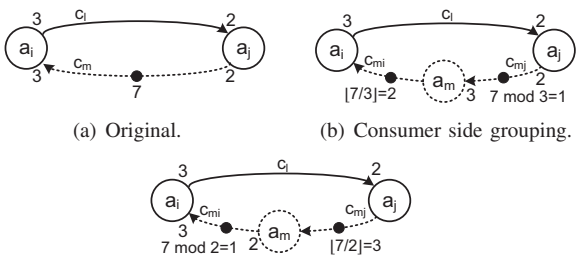
Figure 7. Symbolic MPAG of the example SADF.

symbolic execution. This prevents an explosion in the number of terms in the maximum operation. We construct the symbolic MPAG using the FSM of the SADF and the symbolic Max-Plus matrices $symG_1 \ldots symG_n$ in the same way as the concrete MPAG is constructed in [8]. The only difference is that we use expressions instead of concrete numbers as edge weights. The function getSymMPAG in Algorithm 1 is used to construct the symbolic MPAG *symMPAG*. Fig. 7 shows the symbolic MPAG for our example SADF.

To find the critical cycle, and hence the throughput expression, a maximum cycle mean (MCM) analysis must be performed on the symbolic MPAG. As the parameter point ν_r is known, we evaluate the term of each edge in the symbolic MPAG for the parameter values in ν_r. The function evaluateSymMPAG in Algorithm 1 is used for this purpose. The result of the evaluation for the parameter point $\nu_r : \{p_1 = 2, p_2 = 4\}$ is shown in Fig. 7 as a number that follows the term of the edge. These numbers are used as a weight to the corresponding edges. The evaluation of the parametric SADF for the parameter point ν_r results in the MPAG shown earlier in Fig. 4. Using the MCM analysis algorithm proposed in [18] (function maximumCycleMean in Algorithm 1), the algorithm identifies a critical timing cycle ($criticalCycle$) which in turn is used to determine the throughput expression. The critical timing cycle in our example graph is shown with bold arrows in Fig. 7. The symbolic terms of each edge in the critical cycle are used to determine the throughput expression for the given parameter point (lines 9-11 in the algorithm). In our example, for the given point (i.e., $\nu_r : \{p_1 = 2, p_2 = 4\}$), $periodExp = 6p_1 + 9p_2$ and the throughput expression is $e_r = \frac{2}{6p_1+9p_2}$ (is shown by $MCM_2 = 3p_1 + 4.5p_2$ in Fig. 5). Evaluating this expression for the example parameter point gives the same throughput value ($1/24$) that has been found using the concrete Max-Plus automaton in Sec. II-C. This expression is used as an expression for a throughput region in the divide & conquer technique. Continuing the divide & conquer, different throughput expressions result for the corner points. The symbolic MCM for ν_0 and ν_2 are calculated as follows respectively: $MCM_1 = 5.5p_2$, $MCM_3 = 6p_1{+}2.5p_2$. The hyperplanes $MCM_1 - MCM_2 = 0$ and $MCM_3 - MCM_2 = 0$ divide the parameter space into three throughput regions (see Fig. 5) for the example parameterized SADF.

IV. Model transformation

The number of nodes in a MPAG is equal to the sum of the number of initial tokens in the scenario graphs of all FSM states of the SADF. The complexity of our throughput analysis technique (proposed in Sec. III-D) depends, amongst others, on this number. Initial tokens are often used to model buffer



(a) Original. (b) Consumer side grouping.

(c) Producer side grouping.

Figure 8. Modeling buffer with size 7 for channel c_l.

sizes [11] or schedules in dataflow graphs [19]. Consider as an example a common method to model buffer sizes in the SDF sketched in Fig. 8(a). The dashed channel in Fig. 8(a) models a limit on the buffer size of channel c_l (i.e., the 7 tokens on c_m limit the number of tokens that can be present simultaneously in c_l to 7). Applying this transformation to all channels in a dataflow graph may result in a graph with many initial tokens. Dataflow models that contain implementation aspects such as buffer sizes may therefore contain a considerable amount of initial tokens. As a consequence, the MPAG may become large which in turn may lead to a long run-time for our throughput analysis technique. In this section, we introduce two types of model transformation to reduce the number of initial tokens in the dataflow graph without changing the timing behavior of the actors, i.e., the proposed transformations reduce the number of initial tokens without changing the throughput expressions that are found using our technique. Two token reduction techniques for SDFs are suggested in Sec. IV-A. Sec. IV-B explains under which assumptions and how the proposed token reduction techniques can be applied to SADFs.

A. Token reduction for SDFs

Consider again the SDF shown in Fig. 8(a). After applying our model transformation (explained below), the graph is transformed to the one shown in Fig. 8(b) reducing the number of initial tokens from 7 to 3. In Fig. 8(a), 3 tokens are consumed from c_m in each firing of a_i. So, initial tokens of a channel can be grouped based on the consumer actor's rate (e.g., rate 3). We call this value *grouping factor*. Our model transformation replaces any channel c_m with n initial tokens from some actor a_j with production rate v to some actor a_i with consumption rate w with the following constructs: (1) an actor a_m with zero execution time; (2) a channel c_{mj} with $n \mod w$ initial tokens from a_j to a_m with production rate v and consumption rate w, (3) a channel c_{mi} with $\lfloor n/w \rfloor$ initial tokens from a_m to a_i with production and consumption rate 1. Here, w is the grouping factor.

The proposed graph transformation does not affect the timing behavior of SDFs. In the original graph, actor a_i can fire once whenever at least w tokens exist in channel c_m ($w = 3$ in our example). The n initial tokens that are present on channel c_m all have a corresponding entry in the timestamp vector γ_k, i.e., $\gamma_k(t_1) \cdots \gamma_k(t_n)$ denote the production times of these tokens in the k^{th} iteration. Firing actor a_i consumes w tokens at once from channel c_{mi}. Among this group of w tokens, the token which has the largest timestamp may influence the start time of the actor firing. In other words, $\max\{\gamma_k(t_1) \cdots \gamma_k(t_w)\}$ may influence the first firing of a_i of the $(k+1)^{th}$ iteration of the graph. Understanding this principle enables us to group the n initial tokens as much as possible. We can form $\lfloor n/w \rfloor$ groups of tokens and the remaining (i.e., $n \mod w$) tokens should preserve their individual timestamps.

Grouping initial tokens, on the producer actor side of a channel is also possible. Fig. 8(c) shows the resulting graph after applying initial tokens grouping to the producer side of the channel c_m. In this case, the production rate of actor a_j on channel c_m (i.e., rate 2) specifies the grouping factor.

B. Token reduction for SADFs

The introduced token reduction techniques are also applicable to channels of an SADF where the initial tokens of channels only models intra-scenario dependencies. When tokens on channels model inter-scenario dependencies, the following considerations are required. The token reduction techniques are also applicable to channels of an SADF (which contain inter-scenario token labelings) where the rates on the source and destination side of the channels are identical across all scenarios. In case of different rates for a channel on its destination side for some scenarios, the token grouping on the consumer side cannot preserve the token timestamp information; so, this token reduction cannot be applied. In case of different rates for a channel on its source side for some scenarios, the tokens grouping on the producer side gets limited to the minimum amount of the tokens produced into the channel in one iteration across all scenario iterations. In this situation, the grouping factor is determined based on the greatest common divisor (gcd) of the channel's source side rate in all scenarios. Consider an actor a_j which in each of its firing produces $v_1, \cdots, v_s$ tokens in channel c_m in scenarios $S_1, \cdots, S_s$ respectively. The grouping factor is specified with $gcd(v_1, \cdots, v_s)$; in this way timestamp information of tokens can be preserved and the transformation does not affect the timing behavior of the SADF.

V. Experimental Results

Sec. V-A presents an experimental evaluation of our initial token reduction technique. A comparison between our technique and an existing parametric throughput analysis technique for SDFs is made in Sec. V-B. The performance of our technique on SADFs is evaluated in Sec. V-C. All experiments are performed on an Intel core i7 (3 GHz) running Linux.

A. Impact of initial token reduction

We evaluate our technique on benchmark SDFs with buffer sizes identical to the buffers sizes used in experiments of [12]. The token reduction techniques (i.e., consumer side grouping and producer side grouping introduced in Sec. IV) are applied to the graphs. The second column of Tab. II shows the number of initial tokens in the original SDFs. Using the consumer (Cnsr) side tokens grouping reduces the initial tokens to the amount specified in the third column. Applying the producer (Prdr.) side initial token grouping could further reduce the initial tokens (see the fourth column).

Using the proposed token reduction techniques makes the MPAG-based throughput analysis faster for the buffer-aware graphs; for example in the case of the H.263 decoder (without any token reduction), our parametric throughput analysis lasts $37846ms$. The same analysis for the graph on which consumer

side token reduction is applied requires $10600ms$. The analysis time further reduces to $80ms$ for the graph on which both token reductions are applied. We should also mention that token reductions are performed in a negligible amount of time ($< 1ms$) for all benchmark graphs.

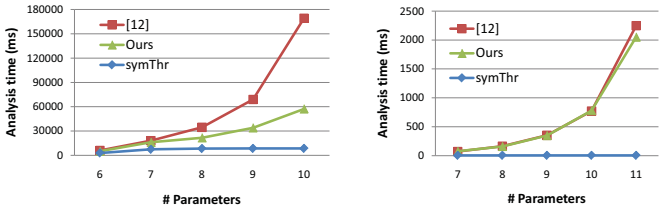
Table II
NUMBER OF INITIAL TOKENS.

Benchmark	Orig.	Cnsr. Opt.	Cnsr.+Prdr. Opt.
H.263 decoder [6]	1193	600	7
H.263 encoder [20]	304	206	10
Modem [1]	54	44	35
MP3 decoder [6]	8078	28	28
MP3 playback [21]	1983	106	106
Samplerate conv. [1]	38	14	14
Satellite receiver [22]	1564	791	48

B. Comparison with [12] on SDFs

Since SDFs are a special case of SADFs, we can compare our technique for SADF throughput analysis to the only existing parametric throughput analysis technique for SDFs, i.e., [12]. In our comparison, we use the same set of SDFs (see Tab. III) with the largest parameter range used in [12]. In this case, the parameterized actors show a variation in their execution time between their nominal value and 150% of this value. Tab. III shows the number of parameters ($\#p$) in each SDF, the number of throughput expressions ($\#eqn$) and the run-times of the technique presented in [12] and our technique. The results show that our technique is faster on almost all SDFs. The MP3 playback and modem SDFs are the only graph on which our technique is slightly slower. This is due to the large number of initial tokens in the graphs which results in a large MPAG and in a long run-time of the MCM algorithm.

The divide & conquer algorithm which is used in both [12] and in our technique performs two different kinds of throughput calculations. It needs to determine throughput expressions for some parameter points and compute concrete throughput values for corner points of throughput regions. The technique from [12] uses a state-space exploration to determine the throughput expressions and concrete throughput values. The state-space exploration needs to keep track of all parameters in its state-space exploration which makes its run-time dependent on the number of parameters. The size of the MPAG used in our technique is not dependent on the number of parameters. Therefore, the run-time of the MCM algorithm is also independent of the number of parameters. Only the function *evaluateSymMPAG* in Algorithm 1 depends on the number of parameters. However this function only evaluates a symbolic MPAG for a parameter point and this can be done in a negligible amount of time. As a result, the run-time of our technique is almost independent of the number of parameters in the graph. The blue line in Fig. 9 shows the run-time of our symbolic throughput computation (i.e., Algorithm 1) when increasing the number of parameters. This result confirms the independence of the run-time of our symbolic throughput computation of the number of parameters. Fig. 9 also compares the overall run-time of the technique from [12] and our technique (including both the symbolic throughput computation as well as all concrete throughput computations at the corner points). When increasing the number of parameters, more concrete throughput computations need to be performed. Our concrete



(a) Satellite receiver. (b) MP3 decoder.

Figure 9. Execution time when varying number of parameters.

throughput analysis technique (which is in principle intended for the more expressive SADF model) is slower than the dedicated SDF technique used in [12]. As a result, part of the gains in our symbolic throughput computation are lost. However, our approach remains faster compared to [12] when increasing the number of parameters. This is especially true when the number of throughput regions increases.

Table III
NUMBER OF THROUGHPUT REGIONS AND RUN-TIME SDFS.

Benchmark	#p	#eqn.	[12] (ms)	Ours (ms)
H.263 decoder [6]	4	1	94	80
H.263 encoder [20]	5	1	36	20
Modem [1]	7	1	96	116
MP3 decoder [6]	8	1	164	160
MP3 playback [21]	1	1	1348	1680
Samplerate conv. [1]	4	2	168	128
Satellite receiver [22]	9	3	69376	33478

Table IV
NUMBER OF THROUGHPUT REGIONS AND RUN-TIME SADFS.

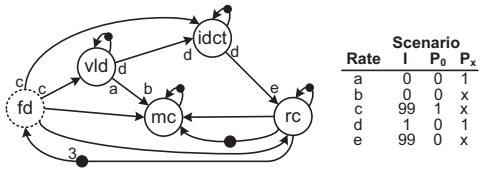
Benchmark	#sce.	#p	#eqn.	Ours (ms)
MPEG4 decoder [23]	9	4	3	488
MP3 decoder [8]	5	8	3	2792
WLAN [24]	4	10	3	2342
Mapped MP3 decoder [8]	5	3	4	1252
Mapped WLAN [24]	4	3	10	196

C. Performance of our technique on SADFs

Our throughput analysis technique is tested on a set of SADFs described in the literature (see Tab. IV). Our benchmark consists of a set of realistic applications, i.e., an MPEG-4 decoder with 9 scenarios, an MP3 decoder with 3 scenarios, and a wireless LAN (WLAN) receiver with 4 scenarios. For the latter two applications, we use both a version in which each actor is mapped to a different processor and a version in which some actors are mapped to the same processor (labeled 'Mapped' in Tab. IV). In the MPEG-4 decoder, each actor is mapped to a different processor. The execution time of all actors is parametrized with a linear expression in which the execution time of the actor in the non-parameterized SADF model is multiplied with a parameter p_i that corresponds to the processor on which the actor is mapped. The actor execution times are varied between their nominal value and 500% of this value (i.e., $1 \leq p_i < 5$). These large parameter ranges allow us to show the scalability of our approach when the parameter ranges are large. Tab. IV shows the number of throughput expressions ($\#eqn$) of each SADF and the time used by our technique to discover these expressions. The experimental results show that our technique is able to handle realistic applications within a limited run-time.

VI. APPLICATION: COMPUTING DVFS SETTINGS

Ref. [23] presents the SADF model of an MPEG-4 video decoder shown in Fig. 10. This model distinguishes 9 different



Rate	Scenario I	P_0	P_x
a	0	0	1
b	0	0	x
c	99	1	x
d	1	0	1
e	99	0	x

Figure 10. SADF model of an MPEG-4 decoder.

scenarios which relate to the type of frame that needs to be decoded (I or P) and the number of macro blocks that need to be processed. The fd and vld actors are mapped to processor p_1 and the three other actors $idct$, mc and rc are mapped to respectively processor p_2, p_3, and p_4. Assuming that an independent, continuous DVFS setting can be made on each processor, the actor execution times as reported in [23] can be scaled linearly with one parameter per processor. Using our parameterized throughput analysis technique, we can compute that the throughput (in iterations/second) of this parameterized SADF is equal to: $\min\{1/(0.00792 \times p_1), 1/(0.003366 \times p_2), 1/(0.00078 \times p_3 + 0.00064 \times p_4)\}$. When using DVFS, the objective is to maximize the parameter values (i.e., minimize processor frequencies) while ensuring that the throughput of the application remains within the throughput constraint. This implies that the minimum of the three throughput expressions of our MPEG4 decoder should not be lower than the throughput constraint of the application. This is a common optimization problem that can be solved with an LP solver. Using such a solver and a throughput constraint of 20 frames/second, we can compute that the processor p_1 in our MPEG-4 decoder should operate at 79 MHz, and all three other processors should operate at 50 MHz.

VII. Related Work

The technique from [12] performs throughput analysis for SDFs when the execution time of the model is parameterized. SADFs are proposed to refine the SDFs for dynamic applications. In [8], a Max-Plus based SADF throughput analysis is introduced. Neither the technique from [8] nor [12] can determine throughput for parameterized SADFs. We propose a technique to enable throughput analysis for parameterized SADFs. Our technique can also be used to determine throughput expressions for parameterized SDFs. Similar to [12], we use a divide & conquer approach to determine throughput regions. In order to determine the throughput expression of a throughput region, we extend the MPAG-based analysis from [8] to a symbolic MPAG-based analysis.

Symbolic executions of the scenario graphs are used in our parametric throughput analysis technique to determine the Max-Plus matrices of an SADF. Symbolic executions may not be practical for large graphs. Approaches like [17], [25] solve this issue by removing redundant expressions which are not affecting the critical paths. In our case, we only need to determine the Max-Plus characteristic matrices for a parameter point. We avoid complex redundant expression elimination by evaluating only the resulting terms (in a token timestamp) for the given parameter point after each symbolic firing of the actors. In this way, only the terms which have the largest values amongst the others propagate to the next step of the symbolic execution. Moreover, the techniques developed in [17], [25], which calculate timing expressions for integrated circuits, are limited to single scenario cases and do not generalize to SADF.

VIII. Conclusions

Scenario-aware dataflow graphs (SADFs) with parameterized execution times enable analysis of implementation decisions that could change the timing property of the application across all scenarios. The only existing parametric throughput analysis technique can handle the less expressive SDF model-of-computation (MoC). Experimental results show that our technique outperforms this technique despite the fact that we can handle a more expressive MoC. Moreover, our technique offers better scalability when the number of parameters increases. As future work, we want to use our technique to develop a timing predictable run-time resource manager.

References

[1] S. S. Bhattacharyya *et al.*, "Synthesis of embedded software from synchronous dataflow specifications," *Journal of VLSI Signal Processing*, vol. 21, pp. 151–166, 1999.

[2] S. Sriram *et al.*, *Embedded Multiprocessors: Scheduling and Synchronization, SecondEdition*. CRC Press, 2009.

[3] P. Poplavko *et al.*, "Task-level timing models for guaranteed performance in multiprocessor networks-on-chip," in *CASES'03*. ACM, pp. 63–72.

[4] M.-Y. Ko *et al.*, "Compact procedural implementation in DSP software synthesis through recursive graph decomposition," in *SCOPES'04*. ACM, pp. 47–61.

[5] A. Ghamarian *et al.*, "Throughput analysis of synchronous data flow graphs," in *ACSD'06*. IEEE, pp. 25–36.

[6] S. Stuijk *et al.*, "Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs," *IEEE Trans. on Computers*, vol. 57, no. 10, pp. 1331–1345, 2008.

[7] B. Theelen *et al.*, "A scenario-aware data flow model for combined long-run average and worst-case performance analysis," in *MEMOCODE'06*, pp. 185 –194.

[8] M. Geilen *et al.*, "Worst-case performance analysis of synchronous dataflow scenarios," in *CODES+ISSS'10*. ACM, pp. 125–134.

[9] A. W. Brekling *et al.*, "Models and formal verification of multiprocessor system-on-chips," *Journal of Logic and Algebraic Programming*, vol. 77, no. 1-2, pp. 1–19, 2008.

[10] I. Sander *et al.*, "System modeling and transformational design refinement in forsyde," *IEEE Trans. on CAD*, vol. 23, pp. 17–32, 2004.

[11] S. Stuijk, "Predictable mapping of streaming applications on multiprocessors," Ph.D. dissertation, TU Eindhoven, 2007.

[12] A. Ghamarian *et al.*, "Parametric throughput analysis of synchronous data flow graphs," in *DATE'08*. EDAA, pp. 116 –121.

[13] S. Bhattacharyya *et al.*, *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.

[14] E. A. Lee *et al.*, "Synchronous data flow," *IEEE Proceedings*, vol. 75, no. 9, pp. 1235 –1245, 1987.

[15] F. Baccelli *et al.*, *Synchronization and Linearity*. John Wiley & Sons.

[16] M. Geilen, "Synchronous dataflow scenarios," *ACM Trans. Embed. Comput. Syst.*, vol. 10, no. 2, pp. 16:1–16:31, Jan. 2011.

[17] K. Heloue *et al.*, "Efficient block-based parameterized timing analysis covering all potentially critical paths," *IEEE Trans. on CAD*, vol. 31, pp. 472–484, 2012.

[18] N. E. Young *et al.*, "Faster parametric shortest path and minimum balance algorithms," *CoRR*, vol. cs.DS/0205041, 2002.

[19] M. Damavandpeyma *et al.*, "Modeling static-order schedules in synchronous dataflow graphs," in *DATE'12*. EDAA, pp. 775–780.

[20] H. Oh *et al.*, "Fractional rate dataflow model for efficient code synthesis," *Journal of VLSI Signal Processing*, vol. 37, pp. 41–51, 2004.

[21] M. H. Wiggers *et al.*, "Efficient computation of buffer capacities for cyclo-static dataflow graphs," in *DAC'07*. ACM, pp. 658–663.

[22] S. Ritz *et al.*, "Scheduling for optimum data memory compaction in block diagram oriented software synthesis," in *ICASSP'95*. IEEE, pp. 2651–2654.

[23] B. Theelen *et al.*, "Scenario-aware dataflow," TU Eindhoven, Tech. Rep. ESR-2008-08, 2008.

[24] O. Moreira, "Temporal analysis and scheduling of hard real-time radios running on a multi-processor," Ph.D. dissertation, TU Eindhoven, 2012.

[25] S. V. Kumar *et al.*, "A framework for block-based timing sensitivity analysis," in *DAC'08*. ACM, pp. 688–693.