# Parametric Throughput Analysis of Synchronous Data Flow Graphs *

A.H. Ghamarian, M.C.W. Geilen, T. Basten, S. Stuijk
Eindhoven University of Technology, Electronic Systems Group
a.h.ghamarian@tue.nl

***Abstract.*** *Synchronous Data Flow Graphs (SDFGs) have proved to be a very successful tool for modeling, analysis and synthesis of multimedia applications targeted at both single- and multiprocessor platforms. One of the most prominent performance constraints of concurrent real-time applications is throughput. For given actor execution times, throughput can be verified by analyzing the SDFG models of such applications, for instance using maximum cycle mean analysis or state space analysis. In various contexts, such as design space exploration or run-time reconfiguration, many fast throughput computations are required for varying actor execution times.*

*We present methods to compute throughput of an SDFG where actor execution times can be parameters. The throughput of these graphs is obtained in the form of a function of these parameters. Recalculation of throughput is then merely an evaluation of this function for specific parameter values, which is much faster than the standard throughput analysis. We propose three different algorithms for parametric throughput analysis and evaluate these algorithms experimentally, showing the feasibility of the approach and showing that a divide and conquer algorithm performs best.*

## 1 Introduction

Synchronous Data Flow Graphs (SDFGs, [10]) are a useful means for modeling and analysis of applications such as DSP applications and concurrent real-time multimedia systems [10, 14, 16, 19]. SDFGs have been used for both single and multiprocessor platforms. The main aim in such systems is realizing a predictable performance. SDFGs are equipped with several timing analysis techniques, which are used for evaluating performance metrics of such applications, most importantly throughput.

An SDFG is a graph where nodes are called *actors* and edges are called *channels*. Actors typically model application tasks. The worst case execution times of tasks are assigned to actors as their execution times and edges model data communication and control dependencies.

Throughput analysis is a crucial indicator of performance used both at design time (e.g., in design space exploration, DSE) and run-time (e.g., resource management). In DSE many different settings of the system are explored [17, 18], which leads to many throughput calculations. At run-time, prediction of throughput is required for proper assignment of resources to applications during reconfigurations [15]. In both cases, throughput calculations need to be as fast as possible with very strict time and resource requirements for run-time applications. Another application example is the study of the impact of variation of execution times under production process variations on throughput [5].

Throughput analysis has been studied in the literature [2, 7] and different methods have been proposed. The execution times of actors are assumed to be fixed numbers in all of the proposed methods. Therefore, any change in the execution time of one or more actors of an SDFG leads to a recomputation of the throughput from scratch. However, calculating the throughput is expensive in many cases, and has exponential time complexity in the worst case.

*This work was supported by the Dutch Science Foundation NWO, project 612.064.206, PROMES.

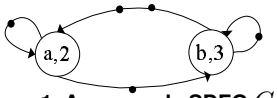


**Figure 1. An example SDFG $G_{ex}$.**

In this paper, we consider parametric SDFGs, a generalization of SDFGs where actors can have parameters as their execution times. We study three algorithms to calculate the throughput of a parametric SDFG as a simple function of the parameters. The resulting function gives the throughput of the SDFG for any value in the range of the parameters. The first two algorithms are variants of the standard throughput analysis algorithms for SDFGs for parametric actor execution times. The third algorithm is based on a divide and conquer (DC) strategy. In the experimental results, we compare the advantages and the drawbacks of these algorithms. The DC algorithm turns out to be the most efficient in practice.

Section 2 introduces synchronous data flow graphs, and their parametric extension. Sections 3, 4 and 5 explain the different methods for finding the throughput of parametric SDFGs. Comparison of the methods is done in Section 6. Section 7 concludes. Proofs can be found in [8].

## 2 Synchronous Data Flow Graphs (SDFGs)

### 2.1 Basic Definitions

SDFGs are a natural means to capture concurrent DSP and multimedia applications. Tasks are modeled as *actors* and channels represent data dependencies. The execution of an actor is referred to as a *firing*, the data items communicated between actors are modeled by *tokens*, and the amounts of tokens produced and consumed in a firing are referred to as *rates*. Channel capacities are unbounded, i.e., channels can contain arbitrarily many tokens (but limited capacities can be modelled in the graph). To make SDFGs amenable to timing analysis, usually fixed execution times are associated to actor firings.

Fig. 1, shows an example of a timed SDFG with two actors, $a$ and $b$, with execution times 2 and 3 respectively, annotated inside the actors. The channel rates in this example are all one. Tokens in channels are shown by black dots.

We assume a set $Ports$ of ports, and with each port $p \in Ports$ we associate a positive finite rate $Rate(p) \in I\!N \setminus \{0\}$. An actor $a$ is a tuple $(In, Out, \tau)$ consisting of a set $In \subseteq Ports$ of input ports ($In(a)$), a set $Out \subseteq Ports$ of output ports ($Out(a)$) with $In \cap Out = \emptyset$ and $\tau \in I\!R$ representing the execution time of $a$ ($\tau(a)$). An SDFG is a tuple $(A, C)$ with a finite set $A$ of actors and a finite set $C \subseteq Ports^2$ of channels. The source of every channel is an output port of some actor; the destination is an input port of some actor. All ports of all actors are connected to precisely one channel.

The execution of an actor is defined in terms of *firings*. When an actor $a$ starts its firing, it removes $Rate(q)$ tokens from all $(p, q) \in C, q \in In(a)$. The firing then continues for $\tau(a)$ time and when it ends, it produces $Rate(p)$ tokens

on every $(p,q) \in C, p \in Out(a)$. An SDFG with all port rates equal to one is called a *Homogenous* Synchronous Data Flow Graph (HSDFG). Any SDFG can be converted [16] to an equivalent HSDFG. However, this conversion may lead to an exponential explosion in the size of the SDFG [13].

Not all SDFGs with arbitrary rates are of practical interest, but only those which comply with an easy to verify condition called *consistency* [10]. Inconsistent graphs either deadlock or are unbounded [6].

## 2.2 Throughput

An SDFG can have more than one execution. Firing every actor as soon as it gets enabled, *self-timed* execution, leads to the highest achievable throughput [16].

**Definition 1** *[Actor Throughput] The throughput $Th(a)$ of an actor $a$ of an SDFG is defined as the average number of firings of $a$ per time unit in the self-timed execution.*

For brevity, we focus on strongly connected SDFGs. The extension of the results to general graphs can be done by combining the results of the strongly connected components of the SDFG [6].

In the literature, there are two different methods for calculating throughput of an SDFG.

**HSDFG method**: It is proven in [11] that the throughput of an SDFG is equal to the inverse of the maximum cycle mean (MCM) of the equivalent HSDFG [16]. The cycle mean $\lambda$ of a cycle of an HSDFG is defined as the total execution time of the cycle over the number of tokens in that cycle. There are efficient algorithms for calculating the MCM of an HSDFG (see [2] for an experimental survey). However, the conversion of an SDFG to an equivalent HSDFG may lead to an exponential explosion in the size of the graph.

**State-space method**: Self-timed execution of an SDFG ends in a repetitive sequence of actor firings, the *periodic phase* of execution. The throughput of an actor can be calculated by dividing the length of the period by the number of firings of the actor in one period.

[7] compares both methods experimentally showing that the state-space method outperforms the HSDFG method.

## 2.3 Parametric SDFGs

A parametric SDFG is an SDFG with at least one actor with a parameter as its execution time. For example, $G_{ex}$ of Fig. 1 becomes a parametric SDFG, $G_{ex}^{par}$, if we assume that the execution times of $a$ and $b$ are given by parameters $p$ and $q$. We are interested in the throughput of a parametric SDFG in the form of a function of the parameters. The domain of this function, the set of values that the parameters can take, is called the *parameter space* of the graph, which is $d$-dimensional when the number of parameters is $d$. Evaluating this function for a point in the parameter space is computationally much cheaper than redoing any of the traditional throughput calculations for those parameter values.

We know that the throughput of an SDFG corresponds to the inverse of the maximum cycle mean of its equivalent HSDFG. The cycle mean of each cycle equals the sum of the execution times of actors in the cycle divided by the number of tokens on the cycle. Consequently, any cycle mean in a parametric HSDFG is a linear combination of parameters with positive coefficients plus a constant, representing the non-parameterized actors in the cycle, which is 0 when there are no such actors. We call these linear combinations *cycle mean expressions*. $1/3p + 1/3q + 0$ is a cycle mean expression of $G_{ex}^{par}$. A cycle mean expression is represented by a vector $\overline{e}$ whose elements are the coefficients of the linear expression, e.g., $(1/3, 1/3, 0)$ in the example. If cycle $c$ has cycle mean expression $\overline{e}_c$, then its cycle mean, $\lambda_c$, for each point $\overline{p} \in I\!R^d$ in the parameter space can be calculated by $\lambda_c(\overline{p}) = \overline{e}_c \cdot (\overline{p}, 1)$, where "$\cdot$" is the inner product of two vectors. For example, for $G_{ex}^{par}$, $\lambda(1,2) = (1/3, 1/3, 0) \cdot (1, 2, 1) = 1$. We denote the evaluation of a point $\overline{p}$ in a cycle mean expression $\overline{e}_c$, by $\overline{e}_c(\overline{p})$. The maximum cycle mean of an SDFG for each point $\overline{p}$ in the parameter spaces denoted by $\lambda^*(\overline{p})$, can be calculated via $\lambda^*(\overline{p}) = \max_{c \in cycles(G)} \overline{e}_c(\overline{p})$.

Note that $\lambda^*$ is a continuous function as it is the composition of continuous functions $\max$ and $\overline{e}_c$. Any expression that has the maximum cycle mean value for some point is called a maximum cycle mean expression (*mcme*). Any mcme that has the maximum cycle mean value for some point for which no other mcme has the maximum value is a *dominating mcme*. It can be shown that the dominating mcmes are sufficient to compute $\lambda^*$. The other cycle mean expressions, including the other mcmes, are called *redundant expressions*.

**Definition 2** *[DCMS] Given an HSDFG, the dominating cycle mean set (DCMS) is the set of dominating mcmes.*

Note that when we talk about the DCMS of an SDFG we refer to the DCMS of its equivalent HSDFG. Conversion of a parametric SDFG to an equivalent HSDFG can be done using the algorithm for non-parametric graphs, since execution times have no impact on the conversion algorithm. We can now express $\lambda^*$ as $\lambda^*(\overline{p}) = \max_{\overline{e} \in DCMS} \overline{e}(\overline{p})$.

Thus, throughput analysis for a parametric SDFG can be done by finding its DCMS. This minimum set can be obtained from the set of all expressions/mcmes by removing all redundant expressions. Checking the redundancy of an expression is equivalent with checking the infeasibility of a linear system [4]. However, since in our case all the coefficients of expressions are positive, there is a fast way to remove a large part of the redundant expressions. An expression is redundant if all of its coefficients are less than or equal to those of another expression. In other words, if we look at the vectors $\overline{e}$ of the expressions, then all points (expressions) are dominated by the pareto set of points and all non-pareto points are redundant expressions. We denote $\overline{e}_1 \preccurlyeq \overline{e}_2$ to express that $\overline{e}_1$ is dominated by $\overline{e}_2$. A pareto dominance test is much easier than the general redundancy checks. Although finding the pareto set of expressions often removes a large part of the redundant expressions, the pareto set is not necessarily the DCMS. For example, suppose our set of expressions is $\{p, q, (p+q)/3\}$, or, in terms of vectors $\{(1,0,0), (0,1,0), (1/3, 1/3, 0)\}$. Even though $(1/3, 1/3, 0)$ is a pareto point (it is not dominated by either $(1,0,0)$ or $(0,1,0)$), there is no point in the parameter space where $(p+q)/3$ has a larger value than all other expressions, making it redundant. Nevertheless, pruning the set of expressions via a pareto dominance test before applying any general redundancy test is worthwhile.

Existing methods of calculating throughput for SDFGs are not directly applicable to parametric SDFGs. The efficient MCM analysis algorithms which work on HSDFGs cannot be applied on parametric SDFGs. The conversion of

SDFGs to HSDFGs can easily be adapted though. Therefore, a naive MCM analysis leads to enumerating all simple cycles of the HSDFG and collecting the expressions in the DCMS. Also, the state-space method of [7] cannot be directly used for parametric throughput analysis, but it can be generalized. In the remainder, we introduce two variations of the existing methods and one new method for calculating the throughput of a parametric SDFG.

# 3 HSDFG Method

This section shows how a parametric throughput can be calculated using the conversion of an SDFG to an HSDFG. The MCM of an HSDFG can be found by enumerating all simple cycles. The cycle mean of each cycle can be calculated by summing up all of the execution times of actors in the cycle and dividing it by the number of tokens on the cycle. Finally, the DCMS of the parametric HSDFG is obtained by removing the redundant expressions as explained in Section 2.3. While enumerating the cycles and calculating their cycle mean expressions only the pareto points are kept.

The example in Fig. 1 is already an HSDFG, so no conversion is needed. This graph has three simple cycles $(a, a)$, $(b, b)$ and $(a, b, a)$ with one, one and three tokens respectively. Therefore, the cycle mean expressions are $p$, $q$ and $(p + q)/3$. Since all expressions are pareto points in the parameter space, no points get eliminated in the pareto test.

In the next step of the algorithm, we see that $(p + q)/3$ is redundant. This follows from the infeasible linear system $\{(p + q)/3 > p, (p + q)/3 > q\}$. Therefore, $DCMS(G) = \{p, q\}$ and $\lambda^*(p, q) = \max\{p, q\}$.

**Algorithm** *HSDFG method*(G)
**Input:** A strongly connected parametric SDFG $G$
**Output:** DCMS of $G$
1. $DCMS = \emptyset$
2. Convert $G$ to equivalent HSDFG $H$
3. **for** each simple cycle $c$ in $H$
4. **do if** $\overline{e}_c \not\preccurlyeq \overline{e}_i$ for all $\overline{e}_i \in DCMS$
5. **then** remove all $\overline{e}_i$ from $DCMS$ for which $\overline{e}_i \preccurlyeq \overline{e}_c$
6. insert $\overline{e}_c$ in $DCMS$
7. Remove redundant expressions from $DCMS$
8. **return** $DCMS$

Note that finding the minimum set of dominating expressions in the whole parameter space (Line 7 of the algorithm) has been solved efficiently in the context of determining the upper envelope of pairwise linear functions [3]. Since the time spent on this part of the algorithm is negligible compared to the first part, we used the straightforward redundancy check explained above in our experiments.

# 4 State-Space Method

State-space-based throughput calculation for SDFGs [7] avoids the conversion to HSDFGs. This section generalizes the state-based method to calculate the throughput of a parametric SDFG.

## 4.1 State Space

The behavior of an SDFG can be defined in terms of a state transition system.

**Definition 3** *[State] The state of a timed SDFG $(A, C)$ is a tuple $(\gamma, \upsilon)$. $\gamma$ associates with each channel the amount of tokens present in that channel in that state. To keep track of time progress actor status $\upsilon : A \to \mathbb{N}^{\mathbb{R}}$ associates with each actor $a \in A$ a multiset of numbers representing the remaining times of different active firings of $a$.*

An actor consumes its required input tokens at the start of its firing, as soon as sufficient tokens are available. Output is produced at the end of the firing. Since channels have infinite capacity, sufficient space is always available. If we are interested in throughput, and not for example in functional analysis, we abstract from the actual communicated data.

In Fig. 1, the initial state is $((1, 1, 1, 2), (\{\}, \{\}))$, where the first vector shows the token distribution of channels, starting from the self-loop channel of $a$ and continuing counterclockwise. The second vector, $\upsilon$, is the vector of multisets of the remaining execution times of $a$ and $b$. Initially, both multisets are empty. At this point, both $a$ and $b$ are enabled and they start their firings, changing the token distribution vector to $(0, 0, 0, 1)$ and the vector of remaining execution times $\upsilon$ to $(\{2\}, \{3\})$. No more actor firings can occur before actor $a$ finishes. So the time goes forward for 2 time units. Completing the firing of $a$ leads to state $((1, 1, 0, 1), (\{\}, \{1\}))$. Firing actor $a$ once again then results in $((0, 1, 0, 0), (\{2\}, \{1\}))$, after which time progresses for 1 time unit, $b$ completes its firing, and so on.

We generalize this model to parametric SDFGs. In the state space of a parametric SDFG, $\upsilon$ contains parametric elements. Since the relations between parameters are not known, we cannot always be sure which firing finishes first. The next example shows how we solve this problem.

The parametric state space of $G_{ex}^{par}$ with execution times $p$ and $q$ for actors $a$ and $b$ is given in Fig. 2. To simplify the figure, the details of states are not shown. Each dot represents a state. The start and end of firings in each state is denoted by the actor name with subscript $s$ or $e$ respectively. For example $a_s$ shows the start of a firing of actor $a$.

After starting firings of $a$ and $b$, $\upsilon$ changes to $(\{p\}, \{q\})$. At this stage, a time step equal to the smallest among all elements in the multisets of $\upsilon$ must be taken, but the relation between $p$ and $q$ is unknown. Therefore, we split the parameter space into two exclusive parts with $p < q$ and $p > q$. For each of these parts, the state space continues in a separate branch. Since our final goal is finding $\lambda^*$ and since $\lambda^*$ is continuous, we do not need to consider the case $p = q$ as the cycle means of this part of the parameter space are covered by expressions obtained by both the cases $p < q$ and $p > q$.

In case $p > q$, the vertical arrow in the figure, after a time step as large as $q$, $b$ finishes its firing. So $\gamma$ and $\upsilon$ become $(0, 0, 1, 2)$ and $(\{p - q\}, \{\})$. The execution proceeds by a time step as large as $p - q$ which leads to the end of the firing of $a$ and consequently the start of new firings of actors $a$ and $b$, changing $\upsilon$ to $(\{p\}, \{q\})$. Since in this branch we already assumed that $p > q$, no new partitioning of the parameter space is needed and the execution proceeds by a time step as large as $q$. The state space in this branch ends in a periodic phase repeating the last two steps. From the periodic phase, we can compute the throughput. The length of the period is $(p-q)+q = p$ and during the period only one firing of actors $a$ and $b$ occurs. Therefore, the throughput is $1/p$ if $p > q$.

We proceed for the case where $p < q$. After a time step as large as $p$, actor $a$ finishes its firing and starts a new firing. So $\gamma$ and $\upsilon$ become $(0, 1, 0, 0)$ and $(\{p\}, \{q - p)\})$ respectively. At this state, the parameter space needs to be split again into two parts: $p < q - p$ and $q - p < p$. Note that the state space only continues if the newly added constraints do not conflict with the previously made assumptions in the earlier states. In this case, both $p < q - p$ and $q - p < p$ are compatible with $p < q$. The case $p < q - p$ $(2p < q)$ gets periodic in a few steps with throughput $1/q$. The other case requires a new

partitioning of the parameter space. Each branch continues till either it ends up in the periodic phase or the constraint set contains conflicting constraints. As shown in the figure the state space of the example continues to repeat a similar pattern. All subsequent branches have the same throughput $1/q$. We can conclude that, as before, $DCMS(G_{ex}^{par}) = \{p, q\}$ for the whole parameter space.

From the example, we can see that the multisets in $\upsilon$ contain linear combinations of parameters throughout the execution of the graph. We also observe that the equations partitioning the parameter space need to be stored in the states.

**Definition 4** *[Parametric State] The state of a parametric SDFG $(A, C)$ is a tuple $(\gamma, \upsilon, \Phi)$. $\gamma$ associates with each channel the amount of tokens present in that channel in that state. Actor status $\upsilon : A \rightarrow \mathbb{N}^T$ associates with each actor $a \in A$ a multiset of linear combinations of actors execution times, each such combination formally denoted by a vector $t$ in $T = \mathbb{Z}^d$ containing the coefficients of the linear expression $t > 0$. The state constraint set $\Phi$ is a subset of $T$, which contains all of the assumptions on the parameters made so far.*

In Fig. 2, the horizontal branches of the state space continue to partition the parameter space into ever smaller pieces. The further splitting of the parameter space does not result in the infeasibility of the system of inequalities. This shows that the state space can be infinite for graphs with parameters that can have real values.

To make the state space finite, we confine the state space, by considering only *integer values* for parameters. Furthermore, we bound the parameter values by lower- and upper-bounds. These two assumptions do not impose any limitation in practice as integers can be as exact as needed by choosing smaller units for execution times. Ranges can also be as wide as required. Choosing the values of parameters from bounded integers make the state space finite.

### 4.2 Throughput Calculation

Algorithm *coverStateSpace* shows the state-space method for the throughput calculation of a parametric SDFG. It receives a parametric SDFG $G$ and an initial state $s = (\gamma, \upsilon, \Phi)$ as arguments and returns the DCMS of $G$. It works recursively in a depth first search fashion, branching the parameter space as explained.

The algorithm uses procedure nextState which accepts parameter expression $t$ (an element of a multiset in $\upsilon$) as a clock step, thereby assuming that $t$ is the minimum time that should elapse before any event can occur. Then, nextState returns the next state $n = (\gamma_n, \upsilon_n, \Phi_n)$ if $\Phi_n$ contains integer solutions within the given parameter bounds. For cases where $\Phi_n$ lacks integer solutions or it has no solution $INT\_INF$ and $INF$ are returned respectively. nextState also marks the current state as a branching state (BS) or a non-branching state depending on whether the parameter space is split.

If the algorithm is invoked for state $s$, for every element $t \in \cup_{a \in A} \upsilon(a)$, a new branch is explored, the procedure nextState is called and the new state $n = (\gamma_n, \upsilon_n, \Phi_n)$ is created. We know that none of the states in the periodic phase can be branching since the constraints in the sets of the recurrent states should be identical. Therefore, the search for the recurrent state only occurs in $nBList$, which stores the non-branching states visited since the last branching state. If $\Phi_n$ is feasible and $n$ is non-branching, then the algorithm checks whether the state is recurrent (has already been visited) by comparing it with already stored non-branching states in $nBList$. If the state is recurrent, the algorithm calculates the cycle mean expressions, by calling calcCMExp. The obtained expression, which is always a dominating mcme, is stored in DCMS. Then the algorithm returns and continues at the last stored branching state if any is left. If $n$ is not recurrent, it is stored in $nBList$ and the algorithm is invoked recursively for $n$.

In case $n$ is a branching state and integer solutions are left, the algorithm is invoked recursively after clearing $nBList$. If $\Phi_n$ is not infeasible but contains no integer solutions, some integer solutions may still exist on the border of $\Phi_n$. Since we only consider strict conditions in $\Phi_n$, we need to account for these points too. For finding mcmes of these points, the *Divide&Conquer* algorithm that is explained in the next section is called. This algorithm finds the mcmes of all points in the region including integer solutions on the border of $\Phi_n$.

The state equivalence check used in the algorithm is a syntactical check except for the constraint sets which are compared based on their integer solutions within the parameter ranges.

**Algorithm** *coverStateSpace*$(s, G)$
**Input:** A parametric state $s = (\gamma, \upsilon, \Phi)$
**Input:** A strongly connected parametric SDFG $G = (A, C)$
**Output:** DCMS of $G$

```
1.  for all t ∈ ∪_{a∈A} υ(a)
2.      do n = (γ_n, υ_n, Φ_n) = nextState(t, s, G);
3.          if (n ≠ INT_INF and not BS)
4.              then if (n ∈ nBList)
5.                       then e = calcCMExp(nBList, n);
6.                            insert(e, DCMS);
7.                       else push(nBList, n);
8.                            coverStateSpace(n, G);
9.              else if (n ≠ INT_INF)
10.                      then reset(nBList);
11.                           coverStateSpace(n, G);
12.             else if (n ≠ INF)
13.                      then Divide&Conquer(G, Φ);
```

## 5 Divide-and-Conquer Method

If we have a closer look at the parts of the parameter space that share the same mcme, we observe that these parts form convex polyhedra.

**Proposition 5** *$\{\overline{p} \in \mathbb{R}^d \mid \lambda^*(\overline{p}) = \overline{e}(\overline{p})\}$ is a convex polyhedron for any mcme $\overline{e}$.*

We call these convex polyhedra *throughput regions*. Fig. 3 shows two throughput regions for the running example, corresponding to $\overline{e}_1 = q$ and $\overline{e}_2 = p$ within a rectangular area between corner points $\overline{v}_1, \ldots, \overline{v}_4$. The following corollary directly follows from Proposition 5.

**Corollary 6** *If for every corner point $\overline{v}$ of an arbitrary polyhedron of the parameter space, $\lambda^*(\overline{v}) = \overline{e}(\overline{v})$ for some mcme $\overline{e}$, then for every point $\overline{p}$ in that region, $\lambda^*(\overline{p}) = \overline{e}(\overline{p})$.*

Hence, the parameter space is composed of throughput regions. Suppose $\overline{e}$ is the mcme of an arbitrary interior point of a convex polyhedron $C$. By comparing the evaluation of $\overline{e}$ for every vertex (corner point) of $C$ and the actual maximum cycle mean value of that vertex, we can detect whether $C$ is a subset of a single throughput region or whether it is covered by parts of more regions. If for any vertex, these two

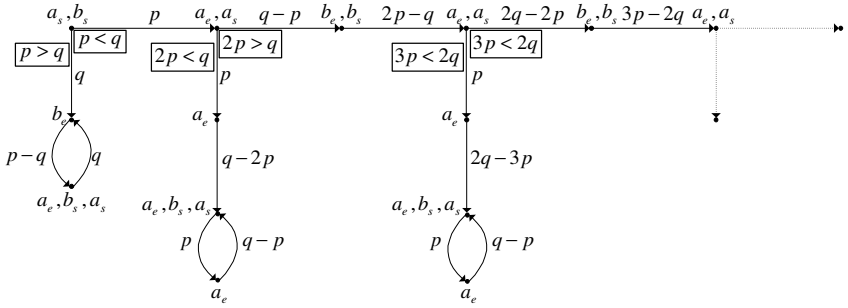


**Figure 2. The parametric execution of $G_{ex}^{par}$.**

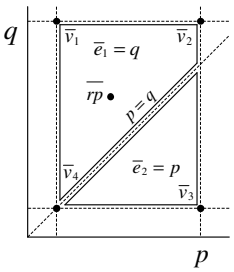


**Figure 3. Divide-and-Conquer Method.**

compared values are different, then $C$ is covered by more than one throughput region; otherwise it is part of a single throughput region (namely that of dominating mcme $\overline{e}$). This idea can be used in a divide-and-conquer method if we add a partitioning strategy, to be applied after detecting a region with more than one mcme. Partitioning continues till all the created regions have a single mcme. All obtained mcmes together form the DCMS.

Since $\lambda^*$ is continuous, the mcmes of two neighboring regions are valid for all points on the border of the regions. This means that for any two neighboring regions with mcmes $\overline{e}_1$ and $\overline{e}_2$, their border is characterized by the equation $\overline{e}_1(\overline{p}) = \overline{e}_2(\overline{p})$. We address this (hyper) plane as the *splitting plane*. In other words, if we have two mcmes for two neighboring regions, we can directly calculate the border of the two regions. The following proposition shows that a splitting plane obtained from two mcmes of a region always passes through the region and splits the region into smaller ones.

**Proposition 7** *Let $\overline{e}_1$ and $\overline{e}_2$ be mcmes associated to points $\overline{p}_1$ and $\overline{p}_2$ respectively. If $\overline{e}_1(\overline{p}_1) \neq \overline{e}_1(\overline{p}_2)$ and $\overline{e}_2(\overline{p}_1) \neq \overline{e}_2(\overline{p}_2)$, then points $\overline{p}_1$ and $\overline{p}_2$ are on opposite sides of the plane characterized by equation $\overline{e}_1(\overline{p}) = \overline{e}_2(\overline{p})$.*

Using Proposition 7 and Corollary 6 we have our complete algorithm if we can find the mcme of a point in the parameter space. This is achieved by adapting the state-space exploration of Section 4. The difference with the generic parametric state-space method is that the evaluation of the expressions in the constraint set are known when searching for an mcme for a concrete point and no branching is required.

Some points in the parameter space may have more than one mcme, if different HSDFG cycles happen to be simultaneously critical. In that case, we may get an expression from this method that does not correspond to any real cycle mean expression of the graph because it contains fragments of different cycles. However, our partitioning strategy only works if the expressions relate to real cycle means. So we need to avoid obtaining such expressions. We show that this is a 'coincidence', that only happens on the border of throughput regions and can be avoided by selecting a random point from the parameter space.

**Proposition 8** *A randomly selected point from the parameter space has only one mcme with probability one.*

Every convex region can be represented in two different ways using half spaces (H-representation) or vertices of the convex region (V-representation) and these two representations are convertible in a very efficient way [4]. In our algorithm, we use both representations, the V-representation for finding the vertices, and the H-representation for calculating the splitting plane. Algorithm *Divide&Conquer*, given below, receives $G$ and a convex region $CR$ as input. Initially, $CR$ is a $d$-dimensional box obtained by the ranges of the parameters. In Line 4 and 5, all the cycle mean values of all the vertices of $CR$ are checked for the validity of the mcme obtained for a random point $\overline{rp}$ in the interior of $CR$. $Th(\overline{v}_i)$ is the throughput of $G$ for point $\overline{v}_i$, obtained using a standard throughput calculation. In case the mcme of $\overline{rp}$ is not valid for a vertex $\overline{v}_i$, then the splitting plane obtained from mcmes of $\overline{v}_i$ and $\overline{rp}$ splits $CR$ (illustrated in Fig. 3 for the example with $\overline{v}_3$ in the role of $\overline{v}_i$) by adding half-spaces characterized by vectors $\overline{e}_{rp} - \overline{e}_{v_i}$ and $\overline{e}_{v_i} - \overline{e}_{rp}$ to $CR$. Then the algorithm is invoked for both subregions in Lines 9 and 10. Procedure ranCornerExpr receives a vertex $\overline{v}_i$, expression $\overline{e}_{rp}$, and $\overline{rp}$. It produces the mcme valid in $\overline{v}_i$ which will be different from $\overline{e}_{rp}$. Note that inside this procedure, instead of using $\overline{v}_i$ itself which is typically on the border between throughput regions, for the reason explained, a randomly selected point in its neighborhood on the line through $\overline{v}_i$ and $\overline{rp}$ is used instead. The algorithm is guaranteed to terminate, because there is a finite number of mcmes and hence a finite number of borders between regions exists that can be used for splitting.

**Algorithm** *Divide&Conquer*($G$, $CR$)
**Input:** A strongly connected parametric SDFG $G$
**Input:** A convex region $CR$
**Output:** DCMS of $G$
1. Let $\overline{rp}$ be a random point in $CR$;
2. $\overline{e}_{rp} \leftarrow findMCME(G, \overline{rp})$;
3. insert($\overline{e}_{rp}$, $DCMS$);
4. **for** all vertices $v_i \in CR$
5. **if** $(\overline{e}_{rp}(\overline{v_i}) \neq 1/Th(v_i))$
6. **then** $\overline{e}_{v_i}$ = ranCornerExpr($v_i, \overline{e}_{rp}, \overline{rp}$);
7. $CR_1 \leftarrow CR \cup \{\overline{e}_{rp} - \overline{e}_{v_i}\}$;
8. $CR_2 \leftarrow CR \cup \{\overline{e}_{v_i} - \overline{e}_{rp}\}$;
9. Divide&Conquer($G$, $CR_1$);
10. Divide&Conquer($G$, $CR_2$);

## 6 Experiments

We have evaluated the execution times of our algorithms using SDFG models of seven real applications. We used the benchmark of [7], consisting of an H.263 decoder, an MP3 decoder, a modem, a satellite receiver, and a sample-rate converter. We further added an H.263 encoder [12] and an MP3 playback application [19]. In *Divide&Conquer*, the CDDLib library [4] is used for all polyhedra operations. In *coverStateSpace*, all operations related to linear inequality systems have been done using LPSolve [1]. All experiments were performed on a P4 PC running at 3.4 Ghz.

**Table 1. Experimental results**

| | #pa | #act | rep | 110% | | | 150% | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | st[s] | dc[s] | #e | st[s] | dc[s] | #e |
| H.263 decoder | 4 | 4 | 1190 | 0.854 | 0.590 | 1 | 0.862 | 0.589 | 1 |
| H.263 encoder | 5 | 5 | 201 | 0.119 | 0.212 | 1 | 0.241 | 0.211 | 1 |
| modem | 7 | 16 | 48 | 51 | 0.568 | 1 | 168 | 0.570 | 1 |
| MP3 decoder | 8 | 14 | 14 | 0.196 | 0.889 | 1 | 0.253 | 0.896 | 1 |
| MP3 playback | 1 | 4 | 10601 | 8.643 | 1.268 | 1 | 17 | 8.177 | 2 |
| sample-rate conv. | 4 | 6 | 612 | 102 | 1.040 | 2 | 266 | 1.246 | 2 |
| satellite rec. | 9 | 23 | 4515 | - | 480 | 3 | - | 450 | 3 |

In each graph, to each actor with varying execution times a parameter has been assigned. Actors with constant execution times received fixed execution times. In cases where more than one copy of an actor existed in the SDFG, the same parameter was dedicated to all copies. Two experiments with the same parameter set and different ranges for the parameters have been carried out. We used two different ranges for parameters with the same lower-bounds and upper-bounds as large as $110\%$ and $150\%$ of the lower-bounds. These ranges were chosen in line with the worst-case estimates of the execution times of the benchmarks, if any were given. The results of these experiments are shown in Table 1 in two different columns. For each experiment, for each graph, the time for both the state-space method (st) and divide-and-conquer (dc) in seconds, as well as the number of expressions in the DCMS (#e) are shown. In all cases, only very few dominating mcmes (up to 3) have been found, which is a good indication for the simplicity of the resulting throughput expression.

The number of parametric execution times (#pa), the number of actors (#act) and the sum of their repetition vector entries (rep, which is also the number of actors in the equivalent HSDFG) of each graph is shown. Since the number of cycles in the equivalent HSDFG directly corresponds to the number of different cycle mean expressions, the sum of repetition vector entries is an important indication for the expected run-time, besides the actor and parameter counts.

We only compared the *Divide&Conquer* and *coverStateSpace* algorithms. The reason is that the *HSDFG method* works on the HSDFGs, and even though we have implemented the fastest cycle enumeration algorithm [9], the algorithm takes generally too long. It only worked for the MP3 decoder for which it only took few a milliseconds to compute the DCMS.

The two methods compared in Table 1 are fast in most cases. The divide-and-conquer method is fast in all the cases. It is also less sensitive to the ranges of parameters than the state-space method. However, its execution time does scale up exponentially with an increasing number of parameters. However, typically, in practical applications only a few parameters are needed since the number of actors with varying execution times is limited and the variations can be captured by he same underlying parameters.

The state-space method works very fast for applications like the H.263 decoder, the H.263 encoder, the MP3 playback and the MP3 decoder, which have a few actors with large execution times. In a few cases, it is faster than divide and conquer. On the other hand, it performs poorly on graphs whose actors have approximately equal execution times. For example, for the satellite receiver, the algorithm took more than a few hours.

Summarizing, the results show that a design-time parametric throughput analysis is feasible. Considering the results in, for example, the context of a run-time resource- or quality management application as proposed e.g. in [15], it is clear that the processing time and memory usage of a throughput calculation for concrete values of the execution time parameters, consisting of an evaluation of the maximum value of the obtained dominating mcmes, are negligible compared to the processing time and memory usage of the typical streaming application. They are in general also small compared to the processing time and memory usage of a traditional throughput calculation, which is typically too expensive to perform at run-time.

# 7 Conclusion

We have extended throughput analysis of SDFGs to parametric SDFGs so that actors can have parameters as their execution times. The throughput of such graphs is a function of the parameters. Evaluating these functions is much faster than traditional throughput analysis methods. We adapted existing methods for computing throughput to parametric SDFGs and proposed a new, faster, algorithm.

# References


[1] M. Berkelaar, K. Eikland, and P. Notebaert. lpsolve. C library, `http://www.geocities.com/lpsolve/`.

[2] A. Dasdan. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *ACM ToDAES*, 9(4):385–418, 2004.

[3] H. Edelsbrunner *et al.* The upper envelope of piecewise linear functions: Algorithms and applications. *Discrete & Computational Geometry*, 4:311–336, 1989.

[4] K. Fukuda. cddlib. C++ library, Swiss Federal Institute of Technology, `http://www.ifor.math.ethz.ch/~fukuda/cdd_home/cdd.html`.

[5] S. Garg *et al.* System-level process variation driven throughput analysis for single and multiple voltage-frequency island designs. In *DATE*, p. 403–408, 2007.

[6] A. H. Ghamarian *et al.* Liveness and boundedness of synchronous data flow graphs. In *FMCAD'06*, p. 68–75. IEEE, 2006.

[7] A. H. Ghamarian *et al.* Throughput analysis of synchronous data flow graphs. In *ACSD'06*, p. 25–36. IEEE, 2006.

[8] A. H. Ghamarian *et al.* Parametric throughput analysis of synchronous data flow graphs. Tech. report ESR-2007-08, TU Eindhoven, http://www.es.ele.tue.nl/esreports/, 2007.

[9] D. B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM J. on Computing*, 4(1):77–84, 1975.

[10] E. Lee *et al.* Synchronous dataflow. *Proc. of the IEEE*, 75(9):1235–1245, September 1987.

[11] R. M. Karp *et al.* Properties of a model for a parallel computations: Determinacy, termination, queueing. *SIAM J. on Applied Mathematics*, 14(6):1390–1411, 1966.

[12] H. Oh *et al.* Fractional rate dataflow model for efficient code synthesis. *J. of VLSI Signal Processing*, 37(1):41–51, 2004.

[13] J. Pino *et al.* A hierarchical multiprocessor scheduling system for DSP applications. In *Conf. on Signals, Systems and Computers*, p. 122–126. IEEE, 1995.

[14] P. Poplavko *et al.* Task-level timing models for guaranteed performance in multiprocessor networks-on-chip. In *CASES'03*, p. 63–72. ACM, 2003.

[15] P. Poplavko *et al.* Execution-time prediction for dynamic streaming applications with task-level parallelism. In *DSD'07*, p. 228–235. IEEE, 2007.

[16] S. Sriram *et al. Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc, NY, 2000.

[17] S. Stuijk. *Predictable Mapping of Streaming Applications on Multiprocessors*. Phd thesis, TU Eindhoven, 2007.

[18] S. Stuijk *et al.* Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *DAC'07*, p. 777–782. ACM, 2007.

[19] M. Wiggers *et al.* Efficient computation of buffer capacities for cyclo-static dataflow graphs. In *DAC 07*, p. 658–663. ACM, 2007.