Throughput Analysis of Synchronous Data Flow Graphs

A.H. Ghamarian[1], M.C.W. Geilen[1], S. Stuijk[1], T. Basten[1], A.J.M. Moonen[1,2], M.J.G. Bekooij[2], B.D. Theelen[1], M.R. Mousavi[1]

[1]Eindhoven University of Technology
[2]Philips Research Laboratories Eindhoven
a.h.ghamarian@tue.nl

Abstract

Synchronous Data Flow Graphs (SDFGs) are a useful tool for modeling and analyzing embedded data flow applications, both in a single processor and a multiprocessing context or for application mapping on platforms. Throughput analysis of these SDFGs is an important step for verifying throughput requirements of concurrent real-time applications, for instance within design-space exploration activities. Analysis of SDFGs can be hard, since the worst-case complexity of analysis algorithms is often high. This is also true for throughput analysis. In particular, many algorithms involve a conversion to another kind of data flow graph, the size of which can be exponentially larger than the size of the original graph. In this paper, we present a method for throughput analysis of SDFGs, based on explicit state-space exploration and we show that the method, despite its worst-case complexity, works well in practice, while existing methods often fail. We demonstrate this by comparing the method with state-of-the-art cycle mean computation algorithms. Moreover, since the state-space exploration method is essentially the same as simulation of the graph, the results of this paper can be easily obtained as a byproduct in existing simulation tools.

1 Introduction

Synchronous Data Flow Graphs (SDFGs, [13]) are widely used in modeling and analyzing data flow applications. They have been used in sequential DSP applications [3, 20]. Recently, they are also used for designing and analyzing concurrent multimedia applications realized using multiprocessor systems-on-chip [16]. The main aim is to realize predictable performance and among the performance indicators, throughput is a prominent one.

Throughput analysis of SDFGs has been extensively studied in literature ([7, 8, 12, 17, 22]). To the best of our knowledge, all throughput analysis approaches suggest algorithms which are based on an analysis of the structure of the graphs. The drawback of these approaches is that they are not directly applicable to SDFGs, but can only be applied on some special sort of SDFGs, namely, Homogeneous SDFGs (HSDFG). Therefore, throughput analysis requires a conversion from an SDFG to an equivalent HSDFG, which is always possible in theory, but frequently leads to a prohibitively large increase in the size of the graph, causing algorithms to fail to produce a result.

In this paper, we propose an alternative method for measuring the throughput of SDFGs. This method, unlike existing methods, works directly on an SDFG, avoiding the costly conversion to HSDFG. The method generates and analyzes the SDFG's dynamic state-space by executing it. Although the number of states that may need to be generated and stored can also be large in the worst case situations, experiments show that the method performs good in practice.

The rest of the paper is organized as follows. The next section discusses related work. In Section 3, we introduce the necessary definitions of SDFGs, define an operational semantics for SDFGs and review SDFGs and equivalent Homogeneous SDFGs. Section 4 discusses throughput of data flow graphs and our method for analyzing throughput. Section 5 places the method in the context of Max-Plus algebra and shows that the results are equivalent to spectral analysis of the Max-Plus equivalent of an SDFG. Section 6 explains our experimental method and presents the results of the experiments, comparing the performance of our method with state-of-the-art throughput analysis through minimum cycle mean algorithms. Section 7 concludes.

2 Related Work

Various data flow models have been proposed in literature, such as computation graphs in [12] or SDFG in [13] among many others. SDFGs are an interesting class, because they are analyzable and still expressive enough to model relevant signal processing applications and platforms [16, 20]. Throughput analysis of (H)SDFGs has been studied extensively [8, 7, 17, 22, 12]. All these studies were focused on analysis of HSDFGs and are applicable to SDFGs only through a conversion as described in [13, 20] to HSDFG. Maximum Cycle Mean (MCM) analysis is then used to determine throughput. Karp proposed an algorithm in [12] which forms the basis for other improved algorithms like [6, 8, 22]. An in-depth comparison of the timing behavior of different MCM (related) algorithms is given in [7, 8]. Behaviour of HSDFGs and their throughput can also be analyzed using Max-Plus algebra [1, 6].

HSDFGs and SDFGs correspond to specific subclasses of Petri nets, namely marked graphs and weighted marked graphs, respectively (when abstracting from the data values that are communicated among the graph nodes). Throughput analysis has been studied extensively in the Petri-net literature as well. In [19], an MCM-related analytical method is presented for marked graphs (HSDFGs), which in [5] is extended to certain specific cases of weighted marked

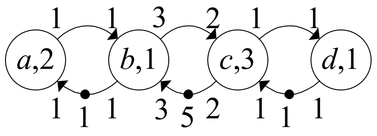

Figure 1. An example SDFG

graphs (SDFGs). In [4], using a linear programming approach, lower and upper bounds on throughput of a certain class of Petri nets are given. The upper bound is exact for marked graphs, and a conversion from weighted marked graphs to marked graphs similar to the conversion of SDFGs to HSDFGs is used to calculate throughput for weighted marked graphs.

Unlike all other previous approaches, we propose a technique based on explicit state-space exploration for finding the throughput which directly works on SDFGs. Because of this, we save the extra step for converting an SDFG to an HSDFG, which can be exponentially larger.

3 Synchronous Data Flow Graphs

In this section, we give a precise definition of Synchronous Data Flow graphs and the notations we use for the purpose of this paper. We introduce a formal, operational semantics of SDFGs and discuss the relation between SDFGs and equivalent homogeneous SDFGs.

3.1 Basic Definitions

Typical DSP and multimedia applications consist of a set of tasks (or operations) that need to be performed in some order, while data is transferred or communicated among those tasks. An important class of DSP and multimedia applications consist of a set of tasks that need to be performed iteratively, while consuming and producing fixed amounts of data for each task execution. This class of applications can be described in a natural way by means of Synchronous Data Flow (SDF) graphs [13]. The nodes of an SDFG are called *actors*, they typically model tasks, while the edges are called *channels*, they typically model data transfers or other dependencies among actors.

The execution of an actor is referred to as an *(actor) firing*, the data items communicated between actors are called *tokens*, and the amounts of tokens produced and consumed in a firing are referred to as *rates*. The classical SDF model is untimed (or in fact it assumes unit time [13]), and actor firings are assumed to be atomic. However, there is a natural extension to SDF in which a fixed execution time is associated with each actor [20]. This extension makes the model amenable to timing analysis, of which throughput analysis is a prominent example.

Example Figure 1 shows an example SDFG, consisting of four actors.The numbers in actor nodes denote their execution times. Associated with the source and destination of each channel edge are the rates. Channels may contain tokens, denoted with a black dot and an attached number defining the number of tokens present in the channel. Channel capacities are unbounded, i.e., channels can contain arbitrarily many tokens. Channel capacity limitations need to be modeled explicitly. For this, the edges in the opposite direction with tokens, model available buffer space. The SDFG of Figure 1 models a multimedia application with four tasks to be executed iteratively in a pipelined manner. The three channels from left to right correspond to FIFO buffers with limited sizes of 1, 5, and 1, resp., as modeled by the channels in the opposite direction.

We assume a set $Ports$ of ports, and with each port $p \in Ports$ we associate a positive finite rate $Rate(p) \in \mathbb{N} \setminus \{0\}$.

Definition 1 *[Actor] An actor a is a tuple (In, Out, τ) consisting of a set $In \subseteq Ports$ of input ports (denoted by $In(a)$), a set $Out \subseteq Ports$ ($Out(a)$) with $In \cap Out = \varnothing$ and $\tau \in \mathbb{N} \setminus \{0\}$ representing the execution time of a ($\tau(a)$).*

Definition 2 *[SDFG] 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, and all channels are connected to ports of some actor. For every actor $a = (I, O, \tau) \in A$, we denote the set of all channels that are connected to ports in I (O) by $InC(a)$ ($OutC(a)$).*

As mentioned, 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 $(p, q) \in InC(a)$. The firing then continues for $\tau(a)$ time units and when it ends, it produces $Rate(p)$ tokens on every $(p, q) \in OutC(a)$. The details of SDFG execution are formalized in the next subsection.

Not all SDFGs are meaningful. Inappropriate rates can lead to undesirable effects. If, for example, in the SDFG of Figure 1, the input rate of actor b of the c-b channel is changed from 3 to 4, this would result in a guaranteed deadlock after only a few actor firings (2 times a, and all other actors once); if this rate is set to 2, it would result in an unbounded increase of tokens in the channel from b to c. There is a simple property, called consistency, of SDFGs that avoids these kinds of effects [13] (although it does not guarantee absence of deadlocks).

Definition 3 *[Consistent SDFG, repetition vector] A repetition vector q of an SDFG (A, C) is a function $A \to \mathbb{N}$ such that for each channel $(i, o) \in C$ from actor $a \in A$ to $b \in A$, $Rate(o) \cdot q(a) = Rate(i) \cdot q(b)$. A repetition vector a is called non-trivial if and only if $q(a) > 0$ for all $a \in A$.*

An SDFG is called consistent *if it has a non-trivial repetition vector. For a consistent graph, there is a unique smallest non-trivial repetition vector, which is designated as the repetition vector of the SDFG.*

The repetition vector of the SDFG of Figure 1 is $\{(a, 2), (b, 2)(c, 3)(d, 3)\}$ (in vector notation: $[2\ 2\ 3\ 3]^T$). The equations $Rate(o) \cdot q(a) = Rate(i) \cdot q(b)$ are called the *balance equations*. From these equations it follows that firing all actors in an SDFG precisely as often as specified by a repetition vector has no net effect on the distribution of tokens over all channels. In the remainder, we always assume consistency. It can be verified very efficiently.

3.2 An Operational Semantics of SDF

We define the behavior of an SDFG formally in terms of a labeled transition system. For this, we need appropriate notions of states and of transitions.

As explained, an actor consumes its required input tokens at the start of its firing, and output is produced at the end of that firing. Channels have infinite capacity, which means that always sufficient space is available. Since we are interested in timing analysis, and not for example in functional analysis, we abstract from the actual data that is being communicated. To capture the timed behavior of an SDFG, we need to keep track of the distribution of tokens over the channels, of the start and end of actor firings, and the progress of time. For distributions of tokens on channels, we define the following concept.

Definition 4 *[Channel quantity] A* channel quantity *on the set C of channels is a mapping $\delta : C \rightarrow \mathbb{N}$. If δ_1 is a channel quantity on C_1 and δ_2 is a channel quantity on C_2 with $C_1 \subseteq C_2$, we write $\delta_1 \preceq \delta_2$ if and only if for every $c \in C_1$, $\delta_1(c) \leq \delta_2(c)$. $\delta_1 + \delta_2$ and $\delta_1 - \delta_2$ are defined by pointwise addition resp. subtraction of δ_1 and δ_2 resp. δ_2 from δ_1; $\delta_1 - \delta_2$ is only defined if $\delta_2 \preceq \delta_1$.*

The amount of tokens read at the beginning of a firing of some actor a can be described by channel quantity $Rd(a) = \{(p, Rate(p)) \mid p \in In(a)\}$, produced tokens by channel quantity $Wr(a) = \{(p, Rate(p)) \mid p \in Out(a)\}$.

Definition 5 *[State] The state of an SDFG (A, C) is a pair (γ, υ). Channel quantity γ associates with each channel the amount of tokens present in that channel in that state. To keep track of time progress, an actor status $\upsilon : A \rightarrow \mathbb{N}^{\mathbb{N}}$ associates with each actor $a \in A$ a multiset of numbers representing the remaining times of different firings of a. The* initial state *of an SDFG is given by some initial token distribution γ. The initial state equals $(\gamma, \{(a, \{\}) \mid a \in A\})$ (with $\{\}$ denoting the empty multiset).*

By using a multiset of numbers to keep track of actor progress instead of a single number, multiple simultaneous firings of the same actor (auto-concurrency) are explicitly allowed. This is in line with the standard semantics for SDF. If desirable, auto-concurrency can be excluded or limited by adding self-loops to actors with a number of initial tokens equal to the desired maximal degree of auto-concurrency.

The dynamic behavior of the SDFG is described by transitions that can be of any of three forms: start of actor firing, end of firing, or time progress with discrete clock ticks.

Definition 6 *[Transitions] A transition of SDFG (A, C) from state (γ_1, υ_1) to state (γ_2, υ_2) is denoted by $(\gamma_1, \upsilon_1) \xrightarrow{\beta} (\gamma_2, \upsilon_2)$ where label $\beta \in (A \times \{start, end\}) \cup \{clk\}$ denotes the type of the transition.*

- *Label $\beta = (a, start)$ corresponds to the firing start of actor a. This transition may occur if $Rd(a) \preceq \gamma_1$ and results in $\gamma_2 = \gamma_1 - Rd(a)$, $\upsilon_2 = \upsilon_1[a \mapsto \upsilon_1(a) \uplus \{\tau(a)\}]$, i.e., υ_1 with the value for a replaced by $\upsilon_1(a) \uplus \{\tau(a)\}$ (where $\uplus$ denotes multiset union).*
- *Label $\beta = (a, end)$ corresponds to the firing end of a. This transition can occur if $0 \in \upsilon_1(a)$ and results in $\gamma_2 = \gamma_1 + Wr(a)$ and $\upsilon_2 = \upsilon_1[a \mapsto \upsilon_1(a) \backslash \{0\}]$ (where $\backslash$ denotes multiset difference).*
- *Label $\beta = clk$ denotes a clock transition. It is enabled if no end transition is enabled and results in $\gamma_2 = \gamma_1$, $\upsilon_2 = \{(a, \upsilon_1(a) \ominus 1) \mid a \in A\}$ with $\upsilon_1(a) \ominus 1$ a multiset of natural numbers containing the elements of $\upsilon_1(a)$ (which are all positive) reduced by one.*

Note that the discrete clock transitions in the above definition can also be replaced by steps in a dense time domain. We confine ourselves to discrete time steps as they are by far the most common in the design of digital systems.

Definition 7 *[Execution] An* execution *of an SDFG is an infinite alternating sequence of states and transitions $s_0 \xrightarrow{\beta_0} s_1 \xrightarrow{\beta_1} \ldots$ starting from the initial state of the graph, such that for all $n \geq 0$, $s_n \xrightarrow{\beta_n} s_{n+1}$.*

Note that even deadlocked executions (when no more actors are firing and no new firings can start) are infinite, because time can always progress. The maximal throughput (a precise definition of throughput is given in the next section) of an SDFG is known to be obtained from one specific type of execution, namely self-timed execution [20], which means that actors must fire as soon as they are enabled.

Definition 8 *[Self-timed execution] An execution is* self-timed *if and only if clock transitions only occur when no start transitions are enabled.*

In the self-timed execution of an SDFG, from one clock transition to the next, there can be some interleaving of simultaneously enabled start and/or end transitions. However, because these start and end transitions are completely independent of each other, independent of the order in which these transitions are applied, the final state before each clock transition, and hence also the state after each clock transition, is always the same. Self-timed SDFG behavior is therefore deterministic in the sense that all the states immediately before and after clock transitions are completely determined and independent of the selected execution.

Example Figure 2 illustrates the self-timed execution of the example SDFG of Figure 1. All clock transitions are shown explicitly. The order of start and end transitions between two clock transitions is irrelevant, and therefore they are conveniently shown as one annotated (macro) step. The example shows that the self-timed execution consists of a periodic phase preceded by a so-called transient phase. We show in the next section that this is always the case for self-timed execution of connected graphs.

Definition 9 *[Iteration] Assume SDFG (A, C) has repetition vector q. An* iteration *is a set of actor firings such that for each $a \in A$, the set contains $q(a)$ firings of a.*

In the execution of Figure 2, the periodic phase with a duration of 12 time units consists of precisely one iteration. In general, it is possible that iterations in an execution of an SDFG overlap in time.

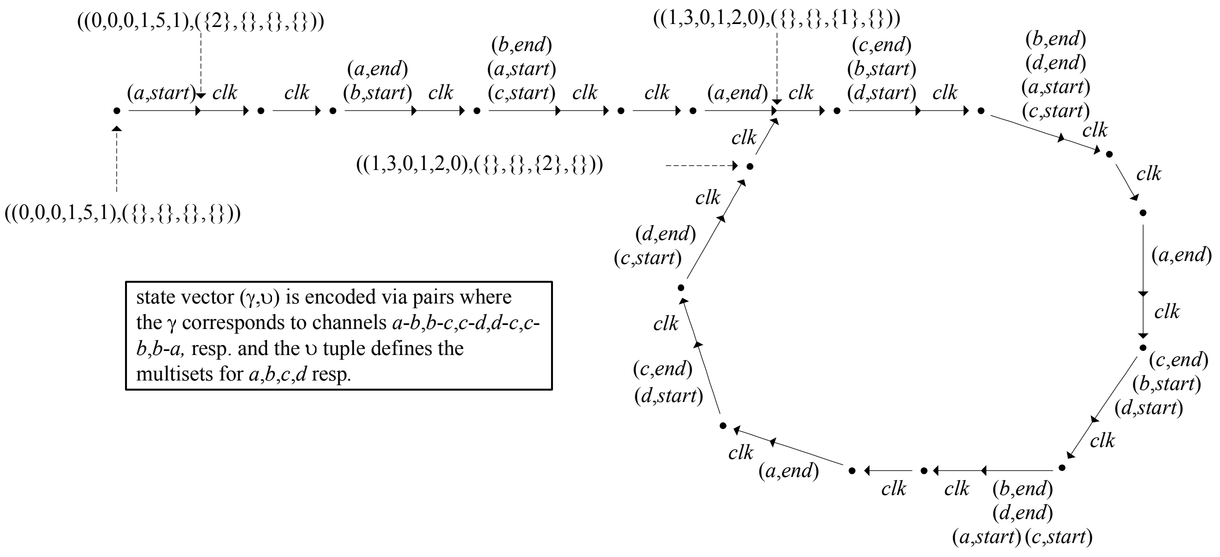

Figure 2. The self-timed execution of our running example

3.3 Homogeneous SDF

SDFGs in which all rates associated to ports equal 1 are called Homogeneous Synchronous Data Flow Graphs (HSDFGs, [13]). As all rates are 1, the repetition vector for an HSDFG associates 1 to all actors.

Every SDFG $G = (A, C)$ can be converted to an equivalent HSDFG $G_H = (A_H, C_H)$ ([13, 20]), by using the conversion algorithm in [20, Section 3.8]. Equivalence in this context means that there exists a one-to-one correspondence between the SDFG and HSDFG actor firings.

Definition 10 *Given an SDFG $G = (A, C)$ and an execution σ, let $t^{\sigma}_{a,k}$ denote the start time of the k-th firing of any actor $a \in A$ in execution σ, i.e., the number of clock transitions up to the the k-th appearance of $\overset{(a,start)}{\rightarrow}$ in σ. If σ is clear from the context, we write $t_{a,k}$.*

For every actor $a \in A$ of an SDFG $G = (A, C)$, with repetition vector q, the conversion algorithm creates $q(a)$ copies, $a_1 \dots a_{q(a)}$, all with execution time $\tau(a)$. The correspondence is as follows: the k-th firing of a_r in the HSDFG corresponds to firing $k \cdot q(a) + r$ of a in the original SDFG. It can be shown ([11]) that for the firing start times of a and its copies, we have that for the self-timed execution, for all $r, k \in \mathbb{N}$ with $0 \leq r < q(a)$,

$$t_{a,k \cdot q(a)+r} = t_{a_r,k} \tag{1}$$

Note that the $k \cdot q(a) + r$-th firing of actor $a \in A$, is also the r-th firing of a in iteration k of G. Also, since actor a and all its copies in the HSDFG have the same execution time, there exist a similar equation for the end times of actor firings in the SDFG and the equivalent HSDFG actor firings.

4 Throughput Analysis of SDF Graphs

In this section, we look at throughput of (executions of) SDFGs. First, we study properties of SDFG state-spaces, then we look at the definition of throughput. We review prevailing methods for throughput analysis of SDFGs and formulate our own approach. In the remainder of this paper, we assume that SDFGs are strongly connected and consistent. For graphs that are not strongly connected, analysis is first done on the strongly connected components and then combined for the whole graph [7]. Moreover, SDFGs that model bounded channels are always strongly connected.

4.1 The Self-timed Execution State-Space

The operational semantics of SDFGs with a self-timed execution policy leads to a state-space of a particular shape, illustrated in Figure 2, the state-space of our example. It consists of a finite sequence of states and transitions (called the *transient phase*), followed by a sequence that is periodically repeated ad infinitum (the *periodic phase*).

Proposition 11 *For every consistent and strongly connected SDFG, the self-timed state-space consists of a transient phase, followed by a periodic phase.*

PROOF Self-timed execution is deterministic if we consider the execution in 'macro steps' from one clock transition to another. Strongly connectedness ensures that every actor depends on tokens from every other actor. This guarantees that there is a bound on the difference in the number of firings of actors, relative to the corresponding entries in the repetition vector. From this it follows that the number of tokens that may accumulate in any channel is bounded and that the amount of auto-concurrency is bounded and only a finite number of copies of an actor can be firing at the same time. Since both the number of simultaneous actor firings and the number of tokens in any channel are bounded, the number of states of an SDFG in self-timed execution is finite. This guarantees that the execution will eventually revisit some state that was visited before, signifying the fact that (because of determinism) the execution is then in the periodic regime. □

Note that if the graph deadlocks, the periodic phase consists of a single clock transition. If we have a closer look at the periodic behavior of the graph, we observe the following.

Proposition 12 *The periodic behavior of an SDFG consists of a whole number (possibly 0) of iterations.*

PROOF A single execution of the periodic behavior has no net effect on the number of tokens in the different channels, because it returns to the same state, which includes information about the tokens in channels. From this, it follows immediately that the number of actor firings (starts and ends) satisfies the SDFG's balance equations and thus must be a multiple of the repetition vector. □

4.2 Throughput

Definition 13 *[Actor throughput] The throughput of an actor a for execution σ of an SDFG is defined as the average number of firings of a per time unit in σ. Since executions are infinite, this average is defined as the following limit:*

$$Th(\sigma, a) = \lim_{t \to \infty} \frac{|\sigma|_a^t}{t}.$$

where $|\sigma|_a^t$ denotes the number of occurrences of the transition $\overset{(a,start)}{\rightarrow}$ up to the t-th clock transition in the execution σ. It is easy to see, that when the execution includes an infinite number of $start$ transitions, then this is equal to

$$Th(\sigma, a) = \lim_{k \to \infty} \frac{k}{t_{a,k}^{\sigma}}.$$

Note that this definition expresses the throughput of an SDFG for a particular execution σ. With $Th(a)$ we denote the throughput of the self-timed execution.

The maximum throughput of an SDFG $G = (A, C)$ is associated with the self-timed execution of G [20, 11], as no actor $a \in A$ can start a firing without having enough tokens in all of its input channels and any delay in start of firing of an actor is of no use in increasing the number of firings of a itself or any other actor in the graph. Hereafter in this paper, we only focus on the throughput associated with the self-timed execution.

Lemma 14 *For every consistent and strongly connected SDFG $G = (A, C)$, the throughput of an actor $a \in A$ is equal to the average number of firings per time unit in the periodic part of the self-timed state-space.*

PROOF Trivial in case of a deadlock. Otherwise, considering the state-space of the self-timed execution of G, we know that there is some K such that for all $i > K$, the i-th firing of a is in the periodic phase. Let $|p|$ and $|p|_a$ respectively, be the number of $\overset{clk}{\rightarrow}$ and $\overset{(a,start)}{\rightarrow}$ transitions in one period. The k-th firing of a, when in the periodic phase, can be decomposed as follows: $k = K + m|p|_a + r$ for some positive m and r. The corresponding time of the start of that firing $T_{a,k} = T + m|p| + T_r$, where T is the time the graph first reached the periodic phase and T_r the time of the r-th firing of a in the period, relative to T. Then,

$$Th(a) = \lim_{k \to \infty} \frac{k}{t_{a,k}} = \lim_{m \to \infty} \frac{K + m|p|_a + r}{T + m|p| + T_r} = \frac{|p|_a}{|p|}.$$

□

Proposition 15 *For a consistent and strongly connected SDFG (A, C) with repetition vector q and actors $a, b \in A$, $Th(a) \cdot q(b) = Th(b) \cdot q(a)$.*

PROOF Trivial in case of deadlock. Otherwise, it follows from the previous lemma and Proposition 12 that $\frac{Th(a)}{Th(b)} = \frac{|p|_a}{|p|_b} = \frac{q(a)}{q(b)}$ □

This proposition means that we can define a normalized notion of (maximal) throughput, independent of any specific actor, and applies to the self-timed execution.

Definition 16 *[SDFG throughput] The throughput of an SDFG $G = (A, C)$ is defined as $Th(G) = \frac{Th(a)}{q(a)}$, for an arbitrary $a \in A$.*

Proposition 15 guarantees that the result is independent of the chosen actor a. From Lemma 14 we know that the throughput of an SDFG can be determined from the periodic part of the state-space.

Corollary 17 *The throughput of an SDFG is equal to the number of actor firings per time unit during one period normalized by the repetition vector. This in turn is equal to the number of iterations executed in one period divided by the duration (number of clk transitions) of one period.*

We are now also able to express the relation between throughput of an SDFG and its equivalent HSDFG.

Theorem 18 *Let G be an SDFG and H the corresponding HSDFG obtained from the conversion algorithm of [20], then $Th(G) = Th(H)$.*

PROOF Trivial in case of deadlock. Otherwise, let a be an actor of G and q the repetition vector of G. For any k we have $i \geq 0$ and $0 \leq r < q(a)$ such that $k = i \cdot q(a) + r$ and

$$Th(G) = \frac{1}{q(a)} \lim_{i \to \infty} \frac{i \cdot q(a) + r}{t_{a,i \cdot q(a)+r}}$$

From the correspondence between the SDFG and HSDFG discussed in Section 3.3, we have that $t_{a,i \cdot q(a)+r} = t_{a_r,i}$

$$Th(G) = \frac{1}{q(a)} \lim_{i \to \infty} \frac{i \cdot q(a) + r}{t_{a_r,i}}$$

$$= \lim_{i \to \infty} \frac{i + r/q(a)}{t_{a_r,i}} = \lim_{i \to \infty} \frac{i}{t_{a_r,i}} = Th(H)$$

□

Example Continuing the example of Figure 2, it can be seen that the periodic phase takes 12 time units and includes one iteration of the graph. Actor d executes 3 times during this period. Hence, the throughput of d equals $3/12 = 1/4$. The normalized throughput of the SDFG itself is $1/12$.

The throughput of the SDFG can be determined from the state space. Often, it is also interesting to determine the critical components, i.e., the actors and channels that are constraining the throughput. These are candidates to improve (speed of an actor or capacity of a channel) if we need to increase throughput. This type of information can also be extracted from the state space, see [21] for an example.

Traditionally (see e.g., [20]), throughput of an SDFG is defined as 1 over the *Maximum Cycle Mean* (MCM) of the corresponding HSDFG. The cycle mean of some cycle of an HSDFG or weighted directed graph in general is defined as the total execution time or total weight of the cycle over the number of tokens or the number of arcs in that cycle for the HSDFG and weighted directed graph respectively. The maximum cycle mean over all cycles in the HSDFG or weighted directed graph is called the MCM of the graph. The MCM can be shown ([17],[20, Lemma 7.3] to be equal to the average time between two firings of any of the HSDFG actors. Given Theorem 1, and the observation that all repetition vector entries of an HSDFG are 1, it is easy to see that Definition 16 of SDFG throughput is the same as the traditional definition of throughput.

Corollary 19 *Let G be an SDFG and H the HSDFG obtained from the conversion of [20], then $Th(G)$ is equal to $1/\mu$ if μ is the MCM of H.*

The suggested method (see, e.g., [20]) for computing throughput of an SDFG is as follows. First, convert the SDFG to an equivalent HSDFG and then compute the throughput on this graph. The throughput of the HSDFG can be computed through an MCM algorithm [12, 8]). In Sections 2.5.3 and 2.5.4 of [1] an approach is described to convert an HSDFG to a weighted directed graph in which each channels contains one token and is annotated with a cost (execution time). The MCM of this graph then equals the throughput of the HSDFG. An alternative method to compute the throughput is the use of an Maximum Cycle Ratio (MCR) algorithm [8]. Each edge in the weighted directed graph for the MCR has a cost (execution time of the producing actor in the HSDFG) and a transit time (number of tokens on the channel in the HSDFG). Efficient algorithms for calculating MCMs/MCRs exist, which are compared in [7]. However, MCM/MCR analysis can only be applied to an HSDFG which is often exponentially larger in size than the original SDFG. This makes the approach as a whole not particularly efficient for SDFG throughput analysis, as our experiments below confirm.

4.3 The State-Space Exploration Method

We propose a method that calculates the throughput of an SDFG by directly executing its self-timed behavior. For our method, we enforce a deterministic order of the interleaving of concurrent transitions corresponding to simultaneous start and end transitions in between clock transitions (see Section 3.2). This has no effect on the throughput, but in this way, the entire state-space becomes deterministic.

In principle, we can execute the SDFG while remembering all states we visit until we detect that we are in the periodic phase when we encounter a state that we have visited before. At that point, by Corollary 17, we can calculate the throughput of the graph by counting, for one period, the number of iterations that were executed and the number of clock transitions. Their quotient is the throughput.

We have to store states to detect the periodic phase, but the lengths of the transient and periodic phases can be fairly long and we may need to store a large number of states. The determinism in the state-space however, allows us to store only selected states. Suppose we pass a state that was visited before, but not stored. We then continue the execution in the same way as the first time, revisiting the same states. We only need to be sure that at least one of the states in the periodic part is actually stored and we will encounter it, detecting the cycle. Knowing from Proposition 12 that the periodic behavior consists of a whole number of iterations we choose to only store one state for every iteration. In this way, the periodic behavior always includes at least one state that is stored. (Except when the graph deadlocks in which case the periodic part consists of only a clk transition and zero iterations, but that is immediately detected when a clock transition remains in the same state.)

We can do this as follows. We pick an arbitrary actor a. Then every iteration includes $q(a)$ start (and end) transitions of a. We choose to store all the states reached immediately after every $q(a)$-th execution of a start transition of a.

Using this method, one can detect the period and also the number of iterations of the period and the length in clock transitions can be easily calculated if we additionally store the number of clock transitions between each two stored states. With this information we can calculate the throughput of an SDFG. In this manner we can significantly decrease the number of states that need to be stored and compared, and consequently the memory and time needed for the algorithm.

Since the method is obtained by some additions (storing and comparing states) to the execution of the behavior of the SDFG, it is relatively simple to integrate the analysis method into existing simulation tools for SDFGs.

5 Max-Plus Algebraic Characterization

A very elegant model to reason about (H)SDFGs is the Max-Plus algebra [1, 6]. Execution of a data flow graph is captured as a linear transformation in a special algebra and linear algebra theory is used to analyze such systems. In particular, spectral analysis is directly related to the throughput analysis problem. In Section 4.2 the relation between throughput of an SDFG and the Maximum Cycle Mean of the equivalent HSDFG is shown. The relation between MCM and Max-Plus algebra is discussed in [1]. In this section we study directly the relation between throughput of SDFGs, our throughput analysis algorithm and Max-Plus algebra. The discussion in this section intends to provide additional insights into the asymptotic behavior of the algorithm. Reading it is however not required to understand the experimental results in the rest of this paper.

We first explain some basics of Max-Plus algebra and then talk about the relation between our state-space exploration method and the Max-Plus formulation of (H)SDFG.

5.1 The Max-Plus model of (H)SDF

In a self-timed execution of an HSDFG, each actor starts a firing when there is at least one token on all of its input channels. The existence of these tokens on the input channels depends in turn on the end of actor firings which provide tokens to the channels. In this way, the start times of each actor firing can be expressed in terms of the start times of certain other actor firings. In this section, we assume an HSDFG (A, C) with initial token distribution γ.

Recall that $t_{a,k}$ denotes the start time of the k-th firing of actor $a \in A$ in the self-timed execution. When it ends, it produces the $k+\gamma(c)$-th token on every channel c connected to one of its output ports. We additionally define $t_{c,k}$ as the time at which the k-th token is produced on channel $c \in C$ (where $t_{c,k} = 0$ for all $0 \leq k < \gamma(c)$, because the initial tokens are already there from the start.) $t_{a,k}$ depends on the availability of tokens on all of its inputs and starts as soon as the last of the required tokens has arrived. The tokens are produced when the actor writing to that channel finishes its firing. From this we derive the following equations for the firing times of actors. For each actor $a \in A$ we have the equations (for all $k \geq 0$):

$$t_{a,k} = \max_{c \in InC(a)} t_{c,k}$$

For each $a \in A$ and channel $c \in OutC(a)$ we have the equations (for all $k \geq \gamma(c)$):

$$t_{c,k} = t_{a,k-\gamma(c)} + \tau(a)$$

Combined, this gives a set of equations in which the k-th firing time of every actor is related to the k-th or earlier firing times of other actors. Through substitution and introduction of auxiliary variables (see [6] for details), this set of equations is converted to a set of difference equations of the form:

$$t_{i,n} = \max_j t_{j,n-1} + \tau_{i,j} \tag{2}$$

where the set of variables $t_{i,n}$ includes the firing times of the actors $t_{a,n}$.

It is convenient to formulate these equations using Max-Plus algebra [1] notation. Max-Plus algebra, like conventional algebra, is defined on real numbers $\mathbb{R}$. In Max-Plus algebra the maximum operator is used in the role of addition and is denoted by $\oplus$, and addition, denoted by $\otimes$, is used instead of multiplication. From this, a linear algebra is obtained and equation (2) can be represented using Max-Plus formulation as follows:

$$t_{i,n} = \bigoplus_j t_{j,n-1} \otimes \tau_{i,j}.$$

This set of sum-of-products equations can be encoded as a matrix equation.

$$\mathbf{t}_n = \mathbf{M}\mathbf{t}_{n-1}.$$

where vector $\mathbf{t}_n$ consists of all $t_{i,n}$. $\mathbf{M}$ is a matrix with the coefficients $\tau_{i,j}$. If $\mathbf{t}_0$ encodes the initial state and initial token distribution, then the sequence $\{\mathbf{t}_k \mid k \geq 0\}$, where $\mathbf{t}_k = \mathbf{M}^k \mathbf{t}_0$ describes the evolution of the graph over time. A special role in this plays the eigenvalue equation

$$\mathbf{M}\mathbf{t} = \lambda \otimes \mathbf{t}.$$

The solution characterizes the graph in its periodic phase. For such a vector $\mathbf{t}$ all execution times of the next iteration $(\mathbf{M}\mathbf{t})$ are equal to the corresponding execution times of the current iteration, shifted by λ units of time. With $\mathbf{t}$ being an eigenvector, the same shift occurs for the next iteration and so on. Hence the behavior is periodic and the corresponding throughput is $1/\lambda$ where λ equals the MCM of the graph. Note that through the connection between firing times in SDFGs and in their corresponding HSDFGs, as discussed in Section 3.3, this model also applies to the execution of SDFGs if we take all firing times of one iteration in a single vector. We use this fact in the next section to model our state-space exploration method.

5.2 A Max-Plus Model of the State-Space Exploration

We now show how computation of throughput with the state-space exploration method can be interpreted as a computation of the eigenvalue of the corresponding matrix. This is akin to the so-called power method for computing the dominant eigenvalue in conventional linear algebra (see e.g., [2]).

The vectors $\mathbf{t}_n$ of the previous section capture the absolute firing times of the actors in the execution of the graph. In the state-space we defined, and our exploration of the state-space, we are not concerned with the absolute firing times, but only relative times, such as remaining execution times of actors. Since we store one state for every iteration, we can build a vector of all actor firing times of an entire iteration, counting relative to the starting time of the particular actor firing used to determine which state is being stored. Assume (without loss of generality) that the time of that actor firing is the first element of the vector: $\mathbf{t}_n(1)$. Define $\mathbf{u}_k$ as the relative version of $\mathbf{t}_k$, by subtracting the first entry from each of the entries, which gives all of the firing times relative to the moment the state was last stored.

$$\mathbf{u}_k = \frac{\mathbf{t}_k}{\mathbf{t}_k(1)}$$

(A division by a scalar t denotes a Max-Plus multiplication $(\otimes)$ with the inverse of t, i.e., a subtraction of t in conventional algebra.) We can then derive the following equation.

$$\mathbf{u}_{k+1} = \frac{\mathbf{t}_{k+1}}{\mathbf{t}_{k+1}(1)} = \frac{\mathbf{M}\mathbf{t}_k}{(\mathbf{M}\mathbf{t}_k)(1)} = \frac{\frac{1}{\mathbf{t}_k(1)}\mathbf{M}\mathbf{t}_k}{\frac{1}{\mathbf{t}_k(1)}(\mathbf{M}\mathbf{t}_k)(1)}$$

$$= \frac{\mathbf{M}\frac{\mathbf{t}_k}{\mathbf{t}_k(1)}}{(\mathbf{M}\frac{\mathbf{t}_k}{\mathbf{t}_k(1)})(1)} = \frac{\mathbf{M}\mathbf{u}_k}{(\mathbf{M}\mathbf{u}_k)(1)}.$$

We now have a recursive equation which characterizes the execution of the state-space exploration method. Similarly, one can show that for any $k \geq 0$ and $d \geq 0$.

$$\mathbf{u}_{k+d} = \frac{\mathbf{M}^d\mathbf{u}_k}{(\mathbf{M}^d\mathbf{u}_k)(1)}.$$

From the fact that this execution ends in a periodic phase, we conclude that there exist m and d such that:

$$\mathbf{u}_{m+d} = \mathbf{u}_m = \frac{\mathbf{M}^d\mathbf{u}_m}{(\mathbf{M}^d\mathbf{u}_m)(1)}.$$

Hence, with $\mu = (\mathbf{M}^d \mathbf{u}_m)(1)$, we have a solution to the eigenvalue equation:

$$\mathbf{M}^d \mathbf{u}_m = \mu \otimes \mathbf{u}_m.$$

Here, μ is the total length of the d iterations in the periodic phase and hence, $\mu = \lambda^d$ ($\mu = d \cdot \lambda$ in common algebra), i.e., d times the eigenvalue λ of $\mathbf{M}$ which is identical to d times the MCM of the equivalent HSDFG which is identical to d divided by the throughput of the SDFG.

6 Experimental Results

6.1 Our SDF Throughput Analysis Tool

This section explains the implementation of the throughput analysis algorithm based on the method proposed in this paper. We developed a tool, called *smart*, which takes an XML description of an SDFG as input and produces the C++ code of a program which does the throughput analysis for the supplied SDFG.

The state of an SDFG consists of a tuple (γ, υ). To implement γ, an array with the size of the number of channels can be used. The function υ associates with each actor a multiset of numbers representing the remaining times of different actor firings of the actors. Instead of using a linked list for each actor, our implementation uses a fixed-size array for each actor a in the SDFG. At position i in the array, the number of actors a which have i time remaining before finishing their execution is stored. The maximum size of the array is determined by the largest execution time of all actors in the SDFG. As a result, each state requires a fixed amount of memory which allows for a run-time efficient implementation of state transitions and state comparisons.

Our algorithm builds the state-space of the graph as outlined in Section 4.3. A recurrent state (i.e., a cycle) must be detected from which the throughput can be computed. States are stored on a stack, indexed using a heap. This heap reduces the number of states which must be compared for equality even further. When a recurrent state is detected, the program computes the throughput from the period.

6.2 Experimental Setup

To the best of our knowledge, all existing techniques to compute the throughput use a conversion to HSDF, followed by MCR analysis or via an additional conversion to a weighted directed graph followed by MCM analysis. Alternatively, spectral analysis of the Max-Plus formulation of the HSDF graph can be used. In [9], Dasdan et. al give an extensive comparison of existing MCM, MCR and spectral analysis algorithms. It shows that Dasdan & Gupta's algorithm (DG) [8] which is a variant of Karp's algorithm [12] and Howard's algorithm (HO) [6] which uses spectral analysis have the smallest running times when tested on a large benchmark. In [7], Dasdan shows that also Young-Tarjan-Orlin's algorithm (YTO) [22] has a very good practical running time. Originally YTO is formulated as a MCM problem, but Dasdan gives pseudo-code for a MCR formulation of the problem. In our experiments, we compare the running times of our state-space exploration method with these state-of-the-art analysis algorithms.

All algorithms are implemented in *smart* for comparison. For the implementation of HO, the source code offered by the authors of [6] is used. An implementation of YTO is available via [14]. It uses the MCM formulation of the problem. We wrote our own implementation of the DG algorithm using the pseudo-code given in [8]. These algorithms the minimum cycle mean of a graph, while throughput analysis requires a maximum cycle mean computation. All implementations are modified to compute this maximum cycle mean. Our comparison requires a conversion from an SDFG to the weighted directed graphs which are input for the MCM algorithms. This conversion consists of two steps. First, an SDFG is converted into an equivalent HSDFG using the algorithm proposed in [20]. Second, the HSDFG is converted into a weighted directed graph using the approach suggested in Sections 2.5.3 and 2.5.4 of [1]. This step requires the computation of the longest path through a graph from each edge with initial tokens to any node in the graph reachable from this edge without using other edges that contain initial tokens.

We measure the running times of each of the two conversion steps and the MCM algorithms individually. These three values per experiment provide insight in the contribution of the different steps to the total running time required for computing SDFG throughput. We also measure the running time of our state-space exploration method. For this approach, the measured running time consists of the time needed for the self-timed execution, storing and comparing states and computing the throughput from the state-space.

6.3 Benchmark

Currently no standard benchmark set of SDFGs exists. (Note that the benchmark used in [7, 9] is a set of directed graphs, and cannot be used for our purposes.) To compare the running times of existing approaches for calculating throughput to our approach, we developed specific sets of test graphs. The first set of graphs in the benchmark are actual DSP and multimedia applications, modeled as SDFGs. From the DSP domain, the set contains a modem [3], a satellite receiver [18] and a sample-rate converter [3], and from the multimedia domain an MP3 decoder and an H.263 decoder. In all graphs, a bound on the storage space of each individual channel is modeled in the graph. Their bounds are set to the minimal storage space required for a non-zero throughput and is computed using the technique from [10].

As a second set of graphs ('Mimic DSP'), the benchmark contains 100 random SDFGs in which actor ports have small rates and the actors have small execution times. These settings for the rates and execution times make the graphs representative for SDFGs of DSP applications.

The practical problem with the existing algorithms for throughput of an SDFG is that the conversion to an HSDFG can lead to an exponential increase in the number of actors [15]. Our approach should not be affected by this problem. To test this hypothesis, the benchmark contains a set ('Large HSDFG') of 100 randomly generated graphs in which the

Table 1. DSP and multimedia applications

	SS	DG	HO	YTO
Modem $[s]$	$1{\cdot}10^{-3}$	$82{\cdot}10^{-3}$	$81{\cdot}10^{-3}$	$81{\cdot}10^{-3}$
Sample rate $[s]$	$1{\cdot}10^{-3}$	> 1800	> 1800	> 1800
Satellite $[s]$	$4{\cdot}10^{-3}$	> 1800	> 1800	> 1800
MP3 decoder $[s]$	$11{\cdot}10^{-3}$	$1{\cdot}10^{-3}$	$1{\cdot}10^{-3}$	$1{\cdot}10^{-3}$
H.263 decoder $[s]$	4	> 1800	> 1800	> 1800

Table 2. Experimental results

	Mimic DSP	Large HSDFG	Long transient
avg #actors (SDFG)	20	13	284
avg #actors (HSDFG)	1008	8166	284
avg length trans. phase	68	208	4486
SDFG to HSDFG conversion			
avg to HSDF $[s]$	$242{\cdot}10^{-3}$	2	–
var to HSDF $[s^2]$	$264{\cdot}10^{-3}$	11	–
#SDFGs not solved	0	10	0
HSDFG to digraph conversion			
avg to digraph $[s]$	$479{\cdot}10^{-3}$	218	$249{\cdot}10^{-3}$
var to digraph $[s^2]$	$17{\cdot}10^{3}$	$95{\cdot}10^{3}$	$160{\cdot}10^{-3}$
#SDFGs not solved	9	44	0
MCM algorithms			
avg MCM (DG)$[s]$	$271{\cdot}10^{-3}$	2	$2{\cdot}10^{-3}$
avg MCM (HO)$[s]$	$1{\cdot}10^{-3}$	$9{\cdot}10^{-3}$	$<1{\cdot}10^{-3}$
avg MCM (YTO)$[s]$	$1{\cdot}10^{-3}$	$8{\cdot}10^{-3}$	$<1{\cdot}10^{-3}$
var MCM (DG)$[s^2]$	$565{\cdot}10^{-3}$	120	$<1{\cdot}10^{-3}$
var MCM (HO)$[s^2]$	$<1{\cdot}10^{-3}$	$<1{\cdot}10^{-3}$	$<1{\cdot}10^{-3}$
var MCM (YTO)$[s^2]$	$<1{\cdot}10^{-3}$	$<1{\cdot}10^{-3}$	$<1{\cdot}10^{-3}$
MCM based throughput analysis			
avg total (DG) $[s]$	48	222	$252{\cdot}10^{-3}$
avg total (HO) $[s]$	48	220	$250{\cdot}10^{-3}$
avg total (YTO) $[s]$	48	220	$250{\cdot}10^{-3}$
var total (DG) $[s^2]$	$17{\cdot}10^{3}$	$97{\cdot}10^{3}$	$<1{\cdot}10^{-3}$
var total (HO) $[s^2]$	$17{\cdot}10^{3}$	$96{\cdot}10^{3}$	$<1{\cdot}10^{-3}$
var total (YTO) $[s^2]$	$17{\cdot}10^{3}$	$96{\cdot}10^{3}$	$<1{\cdot}10^{-3}$
State-space based throughput analysis			
avg total (SS) $[s]$	$<1{\cdot}10^{-3}$	$<1{\cdot}10^{-3}$	$912{\cdot}10^{-3}$
var total (SS) $[s^2]$	$<1{\cdot}10^{-3}$	$<1{\cdot}10^{-3}$	$77{\cdot}10^{-3}$

rates have a large variation (which tends to cause the exponential increase in the conversion) and all actors have equal execution times (this avoids long transient phases).

A potential problem with our approach is that the self-timed execution must first go through the complete transient phase, while the existing MCM algorithms are not affected by this issue. To test the impact of this potential problem on our approach, the benchmark contains a set ('Long transient') of 100 randomly generated SDFGs in which all actors have a large execution time with a small variation. Such SDFGs typically have a transient phase with a large number of clock transitions. Further all ports have a rate of 1, which makes the SDFGs effectively HSDFGs. This avoids an exponential increase in the number of actors during the SDFG to HSDFG conversion, which is also favorable to traditional throughput analysis methods and thus represents the most difficult input for our algorithm.

6.4 Results

Using the three algorithms described in Section 6.2 and our state-space exploration method (SS), we computed the throughput for all SDFGs contained in the four sets of our benchmark. The most important characteristics of the SDFGs in the benchmark are shown in the first three rows of Table 2. The length of the transient phase is measured here in the number of clock transitions. For the MCM algorithms, we measured the running time of the conversion from the SDFG to the HSDFG, the running time of the conversion from the HSDFG to the weighted directed graph and the running time of DG, HO, YTO separately. For the state-space exploration method, we measured the total running time of the algorithm.

For some of the graphs, it was not possible to compute the throughput within 30 minutes using the HO, YTO or DG algorithms. This is caused by the exponential increase in the number of actors when converting an SDFG to an HSDFG. For these graphs, the throughput calculation is stopped and the running times are not taken into account in the results. This provides an optimistic estimate of the real average running time of the existing approaches on the benchmark.

Table 1 shows the measured running times for the real DSP and multimedia applications. The MCM algorithms can only compute the throughput for the MP3 decoder and modem within 30 minutes. They do not complete the HSDFG to weighted directed graph conversion. Our algorithm computes the throughput for all graphs within 4 seconds.

The columns labeled 'Mimic DSP', 'Large HSDFG' and 'Long transient' in Table 2 show the results of our experiments for the corresponding set of SDFGs. For the MCM algorithms, two conversion steps must be performed before the actual MCM analysis can be performed. The section labeled 'SDFG to HSDFG conversion' in Table 2 shows the measured running time for the conversion from an SDFG to an HSDFG. For 10 graphs from the set 'large HSDFG' it was not possible to complete the conversion within 30 minutes. The second step is the conversion from an HSDFG to a weighted directed graph. The results for the step are shown in the section labeled 'HSDFG to digraph conversion' in the Table. A number of graphs fail to finish this step before the time deadline (see row '#SDFGs not solved' of the section 'SDFG to HSDFG conversion').

The measured running times for the MCM algorithms are shown in the section 'MCM algorithms'. The overall required running time using the existing MCM-based approaches is shown in the section 'MCM based throughput analysis' and the running time for our approach is shown in the section 'State-space based throughput analysis'.

We summarize the most important results from the experiments. The results for the set 'mimic DSP' show that our approach has the best average running time with the lowest variance of all four approaches. Our approach solves all problems, while the others did not complete 9 problems due to the conversion to the directed graph. The column 'Large HSDFG' in Table 2 shows the running times for SDFGs with a large increase in the number of actors when going from the SDFG to the HSDFG. The running time of the existing approaches is strongly impacted by this increase and has grown considerable w.r.t. the results in the previous set. In contrast, our running times have the same average and variance as in the previous set. It is further important to note that the set contains 44 SDFGs for which the conversion from the HSDFG to a weighted directed graph cannot be completed. The results for the 'Long transient' set confirm our expectations that SDFGs with a long transient phase impact the running times of our algorithm while not influencing the running times of the other algorithms. However, the running times of our algorithm are on average still below 1 second, which will be acceptable in most situations.

The conversion to a weighted directed graph is required for MCM analysis and often a bottleneck for analysis. However it is not required for MCR analysis. Dasdan gives in [7] a MCR formulation of YTO (YTO-MCR). One can argue however that the running time of the YTO-MCR algorithm will always be larger than the running time of the SDFG to HSDFG conversion (which is still required) plus the running time of the YTO algorithm used in our experiments, the graph used in YTO is never larger than the graph used in YTO-MCR. Therefore we can conclude from the experimental results that also MCR analysis using YTO-MCR will be slower than our state-space exploration method.

Overall, the experiments show that the running time of the existing approaches is greatly impacted by the SDF to HSDF conversion. The results of the experiments on the real applications show also that this problem appears frequently in practice. On the other hand, our method tends to have acceptable running times even if it is confronted with adverse graphs. We observe that our method has on average better run times than the existing MCM approaches and it can compute the throughput of all tested SDGFs within a maximum of 4 seconds while visiting up-to $12 \cdot 10^6$ states, while the MCM approaches fail to produce results on a substantial number of SDGFs.

7 Conclusion

We have introduced a new approach to throughput analysis of Synchronous Data Flow Graphs known also as weighted marked graphs in the Petri-net literature. Existing methods for throughput analysis include a transformation to Homogeneous Data Flow Graphs (marked graphs) and suffer from an exponential blowup in the number of graph nodes, which makes the approaches fail in certain cases. Our approach is based on explicit state-space exploration and avoids the translation to HSDFGs. We have introduced an operational semantics of SDFGs. We have studied properties of the state-space and derived a method for computing throughput based on the state-space. We have shown that the state-space-based definition of throughput corresponds to the classical definitions in terms of Maximum Cycle Mean of the equivalent HSDFG and the eigenvalue of the corresponding Max-Plus matrix equation. Experiments show that our throughput analysis method performs significantly better in practice than existing approaches.

References

[1] F. Baccelli, G. Cohen, G. Olsder, and J.-P. Quadrat. *Synchronization and Linearity* `http://www-rocq.inria.fr/metalau/cohen/SED/book-online.html`. Wiley, 2001.

[2] S. Berberian. *Linear Algebra*. Oxford University Press, 1992.

[3] S. Bhattacharyya, P. Murthy, and E. Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal on VLSI Signal Process. Syst.*, 21(2):151–166, 1999.

[4] J. Campos, G. Chiola, and M. Silva. Ergodicity and throughput bounds for petri nets with unique consistent firing count vector. *IEEE Transactions on Software Engineering*, 17(2):117–125, 1991.

[5] D. Y. Chao, M. Zhou, and D. T. Wang. Multiple weighted marked graphs. *Preprints of 12th IFAC World Congress, Sydney, Australia*, 4:259–263, July 1993.

[6] J. Cochet-Terrasson, G. Cohen, G. Gaubert, and J.-P. Quadrat. Numerical computations of spectral elements in max-plus algebra. In *Int. Conf. on Syst. Structure and Control, Proc.*, pages 667–674. Elsevier, 1998.

[7] A. Dasdan. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *Trans. on Design Automation of Electronic Systems*, 9(4):385–418, 2004.

[8] A. Dasdan and R. Gupta. Faster maximum and minimum mean cycle algorithms for system-performance analysis. *Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 17(10):889–899, 1998.

[9] A. Dasdan, S. Irani, and R. Gupta. Efficient algorithms for optimum cycle mean and optimum cost to time ratio problems. In *Design Automation Conferrence, Proc.*, pages 37–42. ACM, 1999.

[10] M. Geilen, T. Basten, and S. Stuijk. Minimising buffer requirements of synchronous dataflow graphs with model-checking. In *Design Automation Conferrence, Proc.*, pages 819–824. ACM, 2005.

[11] R. Govindarajan and G. R. Gao. Rate-optimal schedule for multi-rate dsp computations. *Journal of VLSI signal processing*, 9:211–235, 1995.

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

[13] E. Lee and D. Messerschmitt. Synchronous dataflow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[14] mmcycle. `http://elib.zib.de/pub/Packages/mathprog/netopt/mmc-info`.

[15] J. Pino, S. Bhattacharyya, and E. Lee. A hierarchical multiprocessor scheduling system for DSP applications. In *Conf. on Signals, Systems and Computers, Proc.*, pages 122–126. IEEE, 1995.

[16] P. Poplavko, T. Basten, M. Bekooij, J. van Meerbergen, and B. Mesman. Task-level timing models for guaranteed performance in multiprocessor networks-on-chip. In *CASES, Proc.*, pages 63–72. ACM, 2003.

[17] R. Reiter. Scheduling parallel computations. *Journal of the ACM*, 15(4):590–599, 1968.

[18] S. Ritz, M. Willems, and H. Meyr. Scheduling for optimum data memory compaction in block diagram oriented software synthesis. In *Int. Conf. on Acoustics, Speech, and Signal Processing, Proc.*, pages 2651–2654. IEEE, 1995.

[19] J. Sifakis. Use of Petri nets for performance evaluation. In *Measuring, modelling and evaluating computer systems Proc.*, pages 75–93. Elsevier Science, 1977.

[20] S. Sriram and S. Bhattacharyya. *Embedded Multiprocessors Scheduling and Synchronization*. Marcel Dekker, Inc, 2000.

[21] S. Stuijk, M. Geilen, and T. Basten. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. To appear in *Design Automation Conferrence, Proc.* ACM, 2006.

[22] N. Young, R. Tarjan, and J. Orlin. Faster parametric shortest path and minimum-balance algorithms. *Networks*, 21(2):205–221, 1991.