A Unified Framework for Throughput Analysis of Streaming Applications under Memory Constraints

Xue-Yang Zhu
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
zxy@ios.ac.cn

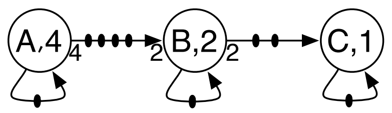
***Abstract*—Streaming applications are an important class of applications in real-time embedded systems, which usually run under restricted resource constraints and with real-time requirement. They are often modeled with Synchronous data flow graphs (SDFGs) or Cyclo-Static data flow graphs (CSDFGs) at the design stage. A proper analysis of the models gives a predictable design for a system. In this paper, we focus on the throughput analysis of (C)SDFGs, taking into account memory constraints. Memory related analysis needs to choose a memory abstraction that decides when the space of consumed data is released and when the required space is claimed. Different memory abstractions may lead to different achievable throughputs. The existing techniques, however, consider only a certain abstraction. If a model is implemented according to other abstractions, the analysis result may not truly evaluate the performance of the system. In this paper, we present a novel unified framework for throughput analysis of memory constrained (C)SDFGs for different abstractions, aiming to provide evaluations matching up to the corresponding implementations. Our methods are exact. Experiments are carried out on several models of real streaming applications and hundreds of synthetic graphs to evaluate the effects and performance of our methods.**

***Keywords*—data flow graphs; iteration period; memory abstractions; self-timed execution; time stamp**

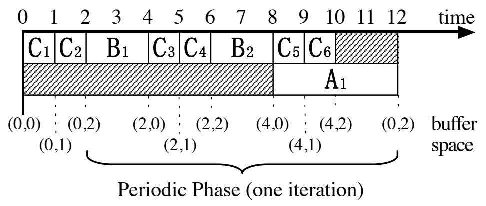
I. Introduction

An important class of applications in embedded systems are streaming applications, such as multimedia and digital signal processing applications. These applications usually run under limited resources and are required to achieve a high throughput.

Synchronous data flow graphs (SDFGs) [1] and *Cyclo-Static data flow graphs* (CSDFGs) [2] are widely used to model streaming applications. Each node (also called *actor*) in an SDFG represents a computation and each edge models a FIFO channel which carries streams of data. A *token* is an atomic data object. A *sample rate* is the number of tokens produced or consumed by per execution of an actor. The sample rates of actors in an SDFG may differ. Execution of all the actors of an SDFG for the required number of times is referred to as an *iteration*, which may include more than one execution, also called a *firing*, of an actor. Different actors of an SDFG may fire a different number of firings in an iteration. The number of firings is decided by the repetition vector of the SDFG, which we will explain later. An example SDFG G_1 is shown in Fig. 1(a). An iteration of G_1 includes one firing of actor A, two firings of B and four of C. The SDFG is a concurrent mode of computation. In an SDFG, there can be simultaneous firings of an actor. This property is called *auto-concurrency*. If auto-concurrency is not allowed for an actor, a self-loop with one initial token can be added. In G_1, for example, auto-concurrency is not allowed. Practical streaming applications modeled with SDFGs include an MP3 playbacker [3], a satellite receiver [4], etc.



(a) The SDFG G_1. The sample rates are omitted when they are 1; the computation time of each actor is attached inside the node; black dots represents initial tokens on the channels. An iteration of G_1 includes one firing of actor A, two firings of B and four of C.



(b) Schedule sch_1 of G_1 with $IP = \frac{1}{thr.} = 10$. The indices of each actor indicate the order of firings of the actor.

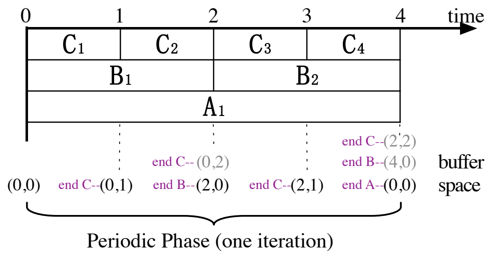

0 1 2 3 4
time
C1 C2 C3 C4
B1 B2
A1
end C--(2,2)
end C--(0,2)
end B--(4,0)
(0,0) end C--(0,1) end B--(2,0) end C--(2,1) end A--(0,0)
buffer space
Periodic Phase (one iteration)
(c) Schedule sch_2 of G_1 with $IP = \frac{1}{thr.} = 4$.

Fig. 1. SDFG G_1 and its schedules under buffer bound $(4, 2)$ corresponding to edges $\langle A, B\rangle$ and $\langle B, C\rangle$.

In an SDFG, the numbers of tokens produced or consumed by different firings of an actor are the same. That is, an actor of an SDFG has fixed sample rates. CSDFGs generalize SDFGs by allowing cyclicly changed sample rates and execution times of an actor. An example CSDFG G_2 is shown in Fig. 2. A channel equalizer is modeled with CSDFG in [5]. A detailed introduction to CSDFGs is in Section III.

This work was supported in part by the National Natural Science Foundation of China (Nos. 61572478, 61472406 and 61472474) and National Key Basic Research Program of China (No. 2014CB340701).

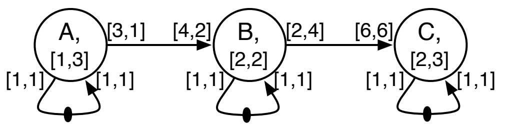


Fig. 2. The CSDFG G_2 [6]. An iteration of G_2 includes six firings of actor A, four firings of B and two of C.

Streaming applications are usually repetitive. Static schedules are typically used to reduce runtime overhead. A *static schedule* arranges the computations of an algorithm to be executed repeatedly. The average computation time per iteration is called the *iteration period* (IP). The IP is the reciprocal of the *throughput*. A smaller IP implies a higher throughput. We will use throughput and IP alternatively in the remained of the paper. When unlimited resources are available, the minimal IP that a (C)SDFG can achieve is decided by its structure and execution times of actors. A graph with recursion (or feedback) have an inherent lower bound on the IP, while the IP of a chain-structured graph can go to infinitesimal, even zero. When any resource is limited, the IP of a (C)SDFG is also restricted by the resource constraints.

In this paper, we are concerned with throughput analysis (also called IP analysis) of SDFGs and CSDFGs, taking into account memory constraints. We try to find the maximal achievable throughputs (i.e. minimal achievable IP) of static schedules of (C)SDFGs. The memories required to implement a (C)SDFG include code and data memory. We focus on the latter. The data flow between actors via channels. Only channel memory is considered in the analysis, meaning that any local buffers needed by the memory abstractions are assumed to be unconstrained. The storage used by channels may be shared or separate. We assume a separate memory.

In the semantics of (C)SDFGs, an actor consumes tokens from its input edges at the start of its firing and produces tokens on its output edges at the end of its firing. When memory constraints are concerned, an analysis needs to choose an abstraction about when to release the buffer space of the consumed tokens and when to claim the buffer space for the produced tokens. Many studies conducted to investigate storage aspects for (C)SDFGs [6], [7], [8] consider only one abstraction, usually the conservative abstraction. They assume that for a firing of an actor, the buffer space needed by the produced tokens is claimed at the start of the firing and the buffer space of consumed tokens is released at the end of the firing. Based on the assumption, a schedule of SDFG G_1, sch_1, is shown in Fig. 1(b). It is under buffer bound $(4, 2)$ corresponding to edges $\langle A, B\rangle$ and $\langle B, C\rangle$, respectively. It arranges firings of actors according to a self-timed execution (STE, also called as soon as possible execution). An STE is a fastest execution of a (C)SDFG and the minimal achievable IP of its corresponding schedule is the average iteration computation time in its periodic phase [9].

The minimal achievable IP of sch_1 is 10. That means no matter how to schedule G_1 under the assumption, its throughput does not exceed 0.1. A clock is usually used in the operational semantic of (C)SDFGs to model the time progress in these methods. At each time point, whether an actor is ready to fire is only decided by the current state, including whether there is sufficient space on its output edges.

This assumption, however, often leads to a pessimistic evaluation of throughputs. Having a close look at sch_1, we find that, at time point 2, after the end of the second firing of actor C, there is space for two tokens on edge $\langle B, C\rangle$, providing sufficient space for the tokens produced by one firing of actor B. This implies that the first firing of B ending at time point 2 will not violate the buffer bound on $\langle B, C\rangle$, if only the second firing of C ends first. Since the tokens on the input edges of B are sufficient for one firing at time point 0 and its execution time is 2, we can arrange it to start one firing at the beginning, concurrently with the first firing of C. Based on this observation, another schedule of G_1 under the same buffer bound, sch_2, is shown in Fig. 1(c). The minimal achievable IP of sch_2 is 4, much better than that of sch_1. Firings in schedule sch_2 are arranged as soon as possible, also forming an STE.

Rather than the abstraction used in schedule sch_1, schedule sch_2 is based on the assumption that the space required by the produced tokens is claimed at the end of the firing. It is obvious that the scheme used in sch_2 leads a faster, or at least not slower, schedule than the scheme of sch_1. An implementation scheduled according to the new scheme may achieve a higher throughput than the analysis result according to the conservative abstraction. The reason is that different memory abstractions may lead to different achievable throughputs of a system.

To match up to the implementation, we present a unified framework for throughput analysis of (C)SDFGs under different memory abstractions. Besides the above two combinations of time abstractions of buffer space claim and release, the framework also deals with other combinations that release the input space at the start of a firing and to claim the output space at the start or the end of a firing. To the best of our knowledge, this is the first work that can provide throughput evaluations under different memory abstractions in a unified framework.

When to analyze the IP of (C)SDFGs based on the above-mentioned new abstraction, two problems arise. When trying to start a firing of an actor, if currently there is no sufficient space on its output edges, how do we know whether the space will be available at the end of the firing? And how to order the start and end events of firings at a same moment? Consider the schedules of G_1 under bound $(4, 2)$ again for example. At time point 0, there are two tokens on $\langle B, C\rangle$, leaving no room for more tokens. However, by the observation we know that at time point 2, there will be enough space on $\langle B, C\rangle$ for the tokens produced by B. It is safe to start the first firing of B at time point 0 and arrange it to end after the end of the second firing of C at time point 2. Then, how to decide the behavior of B at time point 0 according to the state of buffer of $\langle B, C\rangle$ at time point 2, a future state? How do we know that, at time point 2, the second firing of C should end before ending the firing of B; otherwise the buffer bound on $\langle B, C\rangle$ will be violated? The answers to these questions are other two contributions of this paper.

The remainder of this paper is organized as follows. The related work is reviewed in the next section. The main concepts for the proposed methods and the problems addressed are formulated in Sections III. Our main contributions are illustrated in Sections IV and V for SDFGs, and then extended to CSDFGs in Section VI. Section VII provides an experimental evaluation. Section VIII concludes.

II. Related Work

The memories required to implement a (C)SDFG include code and data memory. Code memory minimization is studied in [10]. Studies on the joint minimization of code and data memory for certain classes of SDFGs are conducted in [11], [12]. We focus on the data memory constraints in this paper. The storage used by channels may be shared [13] or separate. We assume a separate memory without buffer merging [14]. Below we review the most related work.

Buffer sizing techniques for minimal storage requirement for a deadlock-free execution are proposed in [15], [16], [17], in which no timing information is considered and therefore no analysis of throughput is provided. We use the minimal storage for a deadlock-free execution of graphs as memory constraints in our experiments.

Techniques proposed in [18], [7], [19] focus on buffer minimization for schedules of SDFGs that achieve the maximal throughput. Govindarajan et al. [18] use the integer linear programming (ILP) technique to solve this problem. Moreira et al. [7] consider a special type of SDFGs, homogenous SDFGs, in which all sample rates of actors are one. Bouakaz et al. [19] use symbolic technique to solve this problem on a special kind of SDFGs - acyclic SDFGs with initially empty channels. Wiggers et al. [3] and Benazouz et al. [20] present heuristics to find minimal buffer capacities of a CSDFG that satisfy a given throughput constraint. The former uses a min-cost network flow formulation and the latter uses the ILP technique. Auto-concurrency is not allowed in the above-mentioned work. Stuijk et al. [6] propose an exact method to determine all Pareto points between throughput and buffer capacities for a (C)SDFG. Auto-concurrency is allowed in (C)SDFGs in this method, but a potential overtaking problem for CSDFGs is not discussed. An STE analysis technique is used in both [7] and [6]. All the above methods are based on the memory abstraction that the space for output tokens is claimed at the start of a firing and the space for input tokens is released at the end of a firing, i.e. the above-mentioned conservative abstraction. The only work that considers minimizing buffer size for optimal throughput based on a different abstraction [21] is proved not exactly minimal [7].

None of them discuss the effect of different abstractions. Our purpose is to provide a unified framework to analyze throughput of a (C)SDFGs under a given memory constraint, based on different abstractions. To the best of our knowledge, there is no work giving such a general solution on this problem.

Our methods are exact and possible to be embedded into other procedures like throughput-buffering trade-off exploration in [6]. We use an STE analysis technique in the solution. Besides the above-mentioned work [6], [7], STE analysis is also used in [9] to compute the maximal throughput of SDFGs, used in [22] to compute the bottleneck of a resource-aware SDFG, in which the resource platform and resource requirement are known. In [23] and [24], it is used to compute throughput after processor allocation for mapping multiple applications. In [8], it is used to find schedules after optimization.

Instead of using a clock in the operational semantics of (C)SDFGs as traditional STE analysis used in above methods, we tag time stamps on tokens to model the time progress. The time stamps of tokens are also used in [25], which defines a Max-Plus linear algebra [26] semantics of SDFGs, to represent the state of a graph after one iteration, and used in [27] to find Pareto optimal schedules with respect to throughput and energy.

III. Preliminaries and problem formulation

In this section, we introduce the main concepts required by the development of our methods and formulate the problems addressed in this paper.

An SDFG is a finite directed graph $G = (V, E)$. V is the set of actors, modeling the computations of a system. Actor v is weighted with its computation time $t(v)$, a nonnegative integer. E is the set of directed edges, modeling interconnections between computations. The source actor and sink actor of e are denoted by $src(e)$ and $snk(e)$, respectively. Edge e is weighted with three properties, $d(e)$, $p(e)$ and $c(e)$. Property $d(e)$ is the number of initial tokens on e, $p(e)$ is the number of tokens produced onto e by each firing of $src(e)$, and $c(e)$ is the number of tokens consumed from e by each firing of $snk(e)$. These numbers are also called the *delay*, *production rate* and *consumption rate*, respectively. The set of incoming edges to actor v is denoted by $InE(v)$, and the set of outgoing edges from v by $OutE(v)$. If $p(e) = c(e) = 1$ for each $e \in E$, G is a *homogenous SDFG* (HSDFG).

A simple SDFG G_1 is depicted in Fig. 1(a). Actor A needs 4 units of time to finish, B needs 2 units and C one unit. The production rate and consumption rate of edge $\langle A, B\rangle$ are 4 and 2, respectively, and there is four initial token on $\langle A, B\rangle$. A firing of actor A will produce 4 tokens on $\langle A, B\rangle$ and actor B can fire only when there are at least 2 tokens on $\langle A, B\rangle$. The actor without any incoming edge is free to fire at any time point. The self-loop on A disallowed auto-concurrency of A. If the self-loop is removed, A is enabled for firing at any time.

SDFG $G = (V, E)$ is *sample rate consistent* [1] if and only if there exists a positive integer vector $q(V)$ satisfying *balance equations*, $q(src(e)) \times p(e) = q(snk(e)) \times c(e)$ for all $e \in E$. The smallest such q is called the *repetition vector*. We use q to represent the repetition vector directly. The repetition vector of G_1 is $q = [1, 2, 4]$, corresponding to actors A, B and C, for example.

In an SDFG, an actor has fixed sample rates and execution time, which usually is the worst case execution time of the computation modeled by the actor. CSDFGs generalize SDFGs by allowing cyclicly changed sample rates and execution times for an actor. Rather than modeling a computation, an actor models a sequence of computations in a CSDFG. Without lose of generality, suppose sequences of all actors have a length of N[1]. A CSDFG $G = (V, E)$ extends above defined SDFG with: there are a sequence of execution times $t(v)[N]$ for each $v \in V$, a sequence of consumption rates $c(e)[N]$ and a sequence of production rates $p(e)[N]$ for each edge $e \in E$. Elements $t(v)[i]$, $c(e)[i]$s for all $e \in InE(v)$ and $p(e)[i]$s for all $e \in OutE(v)$ define the j^{th} firing of v, where $i = j \mod N$. An example CSDFG G_2 is shown in Fig. 2. The first firing of A

[1]According to the original definition of CSDFGs [2], the lengths of sequences may differ from actor to actor. In case there are unequal lengths, a unique N can be defined as the least common multiple of those lengths; or a vector $N(V)$ can be used to model the lengths of sequences of actors.

takes one unit of time to execute and produces 3 tokens on edge $\langle A, B\rangle$; the second firing of A takes 3 units of time to execute and produce 1 token on edge $\langle A, B\rangle$.

The sample rate consistency of a CDFG can be similarly checked and the repetition vector be similarly computed as in SDFGs. The details are referred to [2]. The repetition vector of G_2 is $q = [6, 4, 2]$ corresponding to actors A, B and C, for example.

It is always possible to convert a sample rate consistent (C)SDFG to an equivalent HSDFG [28], [2]. A sample rate consistent (C)SDFG is *deadlock-free* if there is at least one initial token on any cycle in its equivalent HSDFG. Only sample rate consistent and deadlock-free (C)SDFGs are meaningful in practice. Therefore we consider only such (C)SDFGs.

An *iteration* of a (C)SDFG is a firing sequence in which each actor v occurs exactly $q(v)$ times. A *static schedule* arranges the actors of a (C)SDFG to be executed repeatedly. The average computation time per iteration is called the *iteration period* (IP). The *iteration bound* (IB) is the greatest lower bound of the IP for a (C)SDFG with feedback edges.

The IB of an HSDFG is given by its maximum cycle mean [28]. The IB of a (C)SDFG equals the IB of its equivalent HSDFG. Ghamarian et al. [9] and Stuijk et al. [6] present methods to compute the IB of a connected (C)SDFG without converting it to an HSDFG, which we use to evaluate the smallest achievable IPs of memory constrained (C)SDFGs in the proposed framework.

Definition 1. A memory constraint of (C)SDFG $G = (V, E)$ is a vector $MC(E)$, in which $MC(e) \geq d(e)$. When $MC(e) > 0$, it defines the buffer bound of edge $e \in E$; otherwise, e is unbounded.

For convenience, we assume that $MC(e) > 0$ for all $e \in E$, except for self-looped edges. This guarantees a strongly connected MC (C)SDFG defined later.

In the semantics of (C)SDFGs [1], [2], an actor consumes tokens from its input edges at the start of its firing and produces tokens on its output edges at the end of its firing. There are no restrictions on the time when the space of the consumed data is released and when the required space is claimed. When the memory constraints are taken into account in an analysis or scheduling, however, one needs to choose an abstraction about the restrictions.

We use two variables $x \in \{0, 1\}$ and $y \in \{0, 1\}$ to model different abstractions. Variable x is used for the *release abstraction*. The buffer space is released at the start of a firing when $x = 1$, at the end when $x = 0$. Variable y is used for the *claim abstraction*. The buffer space is claimed at the start of a firing when $y = 1$, at the end when $y = 0$. A *memory abstraction* is a combination of release abstraction and claim abstraction, denoted by the value of xy. For example, the above-mentioned conservative abstraction is memory abstraction 01, representing that $x = 0$ and $y = 1$.

Let $IP_{xy}(G, MC)$ be the minimal achievable IP of (C)SDFG $G = (V, E)$ under memory constraint $MC(E)$ based on the memory abstraction xy. The problems addressed in this paper are: given a (C)SDFG G and a memory constraint MC, how to compute $IP_{xy}(G, MC)$ for $x \in \{0, 1\}$ and $y \in \{0, 1\}$.

IV. An Operational Semantics of SDFGs

In this section, we first define an operation semantics of SDFGs without any constraints. Based on the semantics, the self-timed execution (STE) is defined and its properties related to the IP analysis are discussed. The behavior of memory constrained SDFGs is defined in the following section.

Instead of using a clock in the operational semantics of SDFGs as traditional STE analysis methods [6], [9], we use time stamps to model the time progress in a system. Each token tk is tagged with a *time stamp* to record the time when it is produced. Tokens on edge e are expressed by a queue of their time stamps, denoted by $tsE(e)$. All initial tokens have a time stamp 0, indicating that they are available at the beginning. The size of $tsE(e)$ is denoted by $|tsE(e)|$.

We define the behavior of an SDFG G in terms of a *labeled transition system*, represented by $LTS(G)$. A *state* of $LTS(G)$ is the value of $tsE(E)$. The *initial state* of $LTS(G)$ is denoted by s_0. At s_0, for $e \in E$ with $d(e) > 0$, $|tsE(e)| = d(e)$ and the value of each entry in $tsE(e)$ is 0; for e without initial tokens, $tsE(e)$ is an empty queue.

The behavior of an SDFG consists of a sequence of *firings*. We use action $\mathit{fire}(v)$ to model a firing of actor v, and use $\mathit{readyF}(v)$ as predicate capturing its enabling condition.

When there are sufficient tokens on the incoming edges of actor v, it is enabled for a firing. The guard $\mathit{readyF}(v)$ tests if there are sufficient tokens on the incoming edges of v.

$$\mathit{readyF}(v) \equiv_{def} \quad \forall e \in \mathit{InE}(v) : |tsE(e)| \geq c(e).$$

If v has no incoming edges, $\mathit{readyF}(v)$ is always true. That means v is always enabled.

When v is ready to fire, the start time of its firing, $v.st$, is decided by the time stamps of the first $c(e)$ tokens on all incoming edges of v. We are concerned with the STE of SDFGs, so the time is set to be as soon as possible. Since each edge is a FIFO channel, elements in each $tsE(e)$ are always ordered monotonically non-decreasing. Therefore the $c(e)^{th}$ element of $tsE(e)$ is the largest time stamp of the first $c(e)$ tokens on e. The action that chooses the start time of the firing of v is denoted as $sF(v)$.

$$sF(v) \equiv_{def} v.st = \begin{cases} 0, \text{if } \mathit{InE}(v) = \emptyset; \\ \max_{e \in \mathit{InE}(v)} tsE(e)[c(e)], \text{otherwise} \end{cases}$$

Any time later than above defined $v.st$ can be a legal start time of a firing of v, merely it may lead an execution not STE. The end time of the firing is denoted as $v.et$.

$$v.et = v.st + t(v).$$

When a firing of v starts, it consumes tokens of its incoming edges and sets the start time. This action, denoted by $cT(v)$, removes $c(e)$ elements from $tsE(e)$ of each incoming edge e and define $v.st$.

$$cT(v) \equiv_{def} sF(v) \wedge (\forall e \in \mathit{InE}(v) : tsE'(e) = dQ(tsE(e), c(e))),$$

where $tsE'(e)$ refers to the value of $tsE(e)$ in the new state; $dQ(qu, n)$ removes the first n elements from queue qu. The enabling condition $\mathit{readyF}(v)$ guarantees that there are sufficient elements in $tsE(e)$ to be removed.

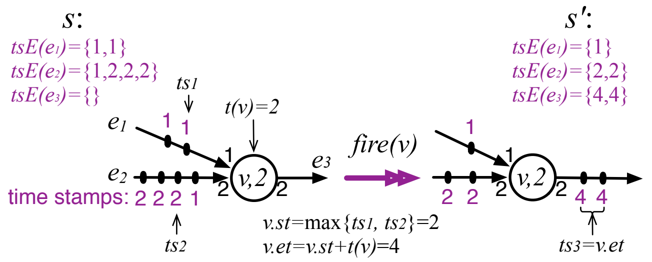


Fig. 3. The effect of a firing of actor v.

When a firing of v ends, it produces tokens to its outgoing edges. The time stamps of those tokens are the time when they are produced, i.e., the end time of the firing. This action, denoted by $pT(v)$, inserts $p(e)$ elements, each with value $v.et$, into $tsE(e)$ of each outgoing edge e.

$$pT(v) \equiv_{def} \forall e \in OutE(v) : tsE'(e) = eQ(tsE(e), p(e), v.et),$$

where $eQ(qu, n, x)$ inserts n xs into queue qu.

Then the behavior of a firing of v is defined as the combination of the token consumption and the token production.

$$fire(v) \equiv_{def} cT(v) \wedge pT(v).$$

The effect of $fire(v)$ is illustrated in Fig. 3. At state s, there are sufficient tokens on e_1 and e_2, the incoming edges of v, so $readyF(v) = true$. The consumption rate of e_1 is 1 and the time stamp of the first token on e_1 is $ts_1 = 1$; the consumption rate of e_2 is 2 and the time stamp of the second token on e_2 is $ts_2 = 2$. That means that the earliest time when v can fire is 2. By action $sF(v)$, $v.st = 2$. Therefore a firing of v starts at time point 2, executing for 2 units of time, and then ends and produces two tokens with time stamp 4 on the outgoing edge e_3. The state is changed to s'.

Action *fire* is the only kind of *action* of $LTS(G)$. A *transition* from state to state of $LTS(G)$ is caused by *fire* constrained by its enabling condition $readyF$. An *execution* of an SDFG G is an infinite alternating sequence of states and transitions of $LTS(G)$. We use actions to represent transitions that are caused by them. By the way that the start time of a firing is set, the execution is an STE.

For a strongly connected SDFG, the numbers of initial tokens, the execution times and the sample rates are finite. Therefore, although the time stamps in an STE may go infinitely, the changes of them are finite. We define *normalized states* to capture the finite changes. Let $||s||$ denotes the *base time stamp* of state s, defined by the time stamp of the earliest produced token at s.

$$||s|| = \min_{ts \in s.tsE(e) \wedge e \in E} ts$$

Denote $Nor(s)$ as normalized state of s. Let each time stamp in s minus its base time stamp to form $Nor(s)$.

$$Nor(s).tsE = s.tsE - ||s||, \text{ where}$$

$$tsE - ||s|| \equiv_{def} \forall ts \in tsE(e), e \in E : ts' = ts - ||s||.$$

An STE σ ultimately goes into a repetitive pattern (called the *periodic phase*) in terms of the normalized states. This property is tested by predicate $hasCycle(\sigma)$. Algorithm 1 returns the STE of a strongly connected SDFG. The periodic phase consists of a whole number of iterations. The average iteration computation time in the periodic phase is exactly the IB of the SDFG [9].

Algorithm 1 STE(G)

```
Require: A strongly connected SDFG $G$
Ensure: The STE $\sigma$ of $G$
 1: $lts = LTS(G)$
 2: $s = lts.s_0$
 3: $\sigma \leftarrow s$
 4: while not $hasCycle(\sigma)$ do
 5:     for all $v \in G$ do
 6:         while $readyF(v)$ do
 7:             $fire(v)$
 8:             $\sigma \leftarrow \sigma + fire(v)$
 9:         end while
10:     end for
11:     $\sigma \leftarrow \sigma + s$
12: end while
13: return $\sigma$
```

Let s_b and s_e denote the beginning state and the end state of the periodic phase of the STE, respectively. They could be any pair of states with $Nor(s_b) = Nor(s_e)$. The time elapsing between s_b and s_e is the computation time of the periodic phase. It can be computed by the base time stamps of the two states. Suppose the periodic phase includes n iterations. The IB of SDFG G can be computed from them. That is,

$$IB(G) = (||s_e|| - ||s_b||)/n. \tag{1}$$

The IB of a deadlock-free strongly connected SDFG is a positive. The IB of an SDFG not strongly connected can be computed by their strongly connected components. We will use the properties of STE to analysis the IP of memory constrained SDFGs. The IB of some memory constrained graphs, which are originally chain-structured graph, may be zero base on certain memory abstraction. The reason is that the semantics of a memory constraint SDFG is defined slightly different from the above defined semantics.

V. Throughput Analysis of Memory Constrained SDFGs

A. Memory-Constrained SDFGs

The bound on buffer of an edge $\langle u, v\rangle$ of an SDFG can be modeled by adding edge $\langle v, u\rangle$ with tokens to model available storage space [6]. Scheduling an SDFG under memory constraint is equivalent to scheduling the corresponding memory-constrained SDFG.

Definition 2. A memory-constrained SDFG (MC SDFG) of $G = (V, E)$ under $MC(E)$ is an SDFG $G_{MC} = (V, E \cup E_{MC})$, in which $E_{MC} = \{\langle v, u\rangle | \langle u, v\rangle \in E \wedge MC(\langle u, v\rangle) > 0\}$. For all $e' = \langle v, u\rangle \in E_{MC}$, there is an edge $e = \langle u, v\rangle \in E$, such that $p(e') = c(e)$, $c(e') = p(e)$ and $d(e') = MC(e) - d(e)$.

Edge $e' \in E_{MC}$ is called *MC edge* of corresponding $e \in E$. Denote $InE_{MC}(v)$ as the set of MC edges of $OutE(v)$ and $OutE_{MC}(v)$ as the set of MC edges of $InE(v)$. An MC SDFG

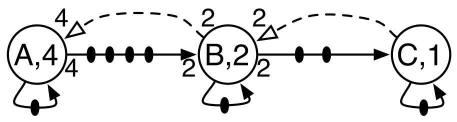


Fig. 4. MC SDFG of G_1 under $MC = (4, 2)$, corresponding to edges $\langle A, B\rangle$ and $\langle B, C\rangle$. MC edges are depicted as dotted lines.

of G_1 (Fig. 1(a)) under $MC = (4, 2)$ is shown in Fig. 4. MC edges are depicted as dotted lines. Since there are already 4 and 2 tokens on edges $\langle A, B\rangle$ and $\langle B, C\rangle$, respectively, no tokens, which model buffer space, available on their corresponding MC edges. An MC SDFG is a strongly connected SDFG. Its minimal achievable IP can be computed exactly by the periodic phase of its STE.

B. Semantics and IP Computation of MC SDFGs

The behavior of G_{MC} now is not only decided by the state of edges in E as defined in Section IV, but also constrained by the space modeled by the tokens on MC edges. The state of its LTS is the value of vector $tsE(E \cup E_{MC})$. The behavior of G_{MC} under memory abstraction xy is defined by enabling condition $readyF_{MC}$, which is not affected by the abstractions, and action $fire_{xy}$, which may lead different states refer to different values of xy. Below let's see how the behavior of an SDFG is restricted by a memory constraint under different abstractions.

The guard $readyF_{sp}(v)$ tests if there are sufficient tokens available on the MC edges of outgoing edges of actor v. If there are no memory constraints on all the outgoing edges, $readyF_{sp}(v)$ is always $true$.

$$readyF_{sp}(v) \equiv_{def} \forall e \in InE_{MC}(v) : |tsE(e)| \geq c(e).$$

When there are sufficient tokens on its incoming edges and on the MC edges of its outgoing edges, actor v is enabled for a firing.

$$readyF_{MC}(v) \equiv_{def} readyF(v) \wedge readyF_{sp}(v)$$

At state s in Fig. 5, there are sufficient tokens on the incoming edges of actor B. That is, $d(\langle A, B\rangle) \geq c(\langle A, B\rangle)$ and $d(\langle B, B\rangle) \geq c(\langle B, B\rangle)$, implying that $readyF(B) = true$;and $d(\langle C, B\rangle) \geq c(\langle C, B\rangle)$, implying that $readyF_{sp}(B) = true$. Therefore actor B is ready for a firing.

Let cTT be the start time of a firing of actor v decided by the tokens on the incoming edges, i.e., $v.st$ assigned by action $sF(v)$ in Section IV. The start time of the firing now is also decided by the time stamps of tokens on the MC edges of the outgoing edges. Let $cTS_y(v)$ be the start time point decided by the MC edges of v, under claim abstraction y. Then the start time of v under memory constraint is the larger one of cTT and $cTS_y(v)$. Let $sT(v)$ be the largest time stamps of the required tokens on MC edges. Action $sF_y(v)$ defined the start time of firing of v.

$$sF_y(v) \equiv_{def} v.st = \max\{cTT, cTS_y\},$$

where

$$cTS_y(v) = \begin{cases} sT(v), & \text{if } y = 1, \\ sT(v) - t(v), & \text{if } y = 0, \end{cases}$$

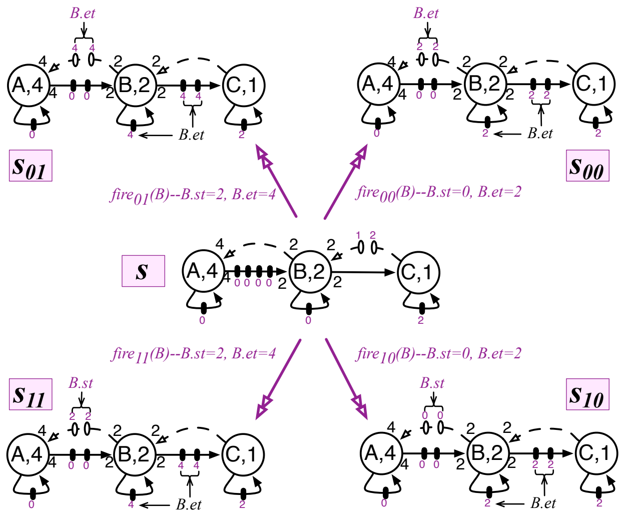

B.et
B.et
B.st
B.st
S01
S00
S
S11
S10
fire01(B)--B.st=2, B.et=4
fire00(B)--B.st=0, B.et=2
fire11(B)--B.st=2, B.et=4
fire10(B)--B.st=0, B.et=2
A,4
B,2
C,1
x=1, release space at the start of a firing; x=0, release at the end;
y=1, claim space at the start of a firing; y=0, claim at the end.

Fig. 5. The effects of $fire_{xy}$.

and

$$sT(v) = \begin{cases} 0, \text{ if } InE_{MC}(v) = \emptyset; \\ \max_{e \in InE_{MC}(v)} tsE(e)[c(e)], \text{otherwise} \end{cases}$$

A firing claims the space it requires at the time according to the claim abstraction y. If the space is claimed at the start of a firing, i.e. $y = 1$, $cTS_y(v)$ is set as $sT(v)$. The firing of v is not allow to start before $sT(v)$. If the space is claimed at the end of a firing, $sT(v)$ can be chosen as the end time of the firing. It is safe to let v start at $sT(v) - t(v)$. Therefore $cTS_0(v) = sT(v) - t(v)$.

The action to claim required space according to claim abstraction y is defined as cS_y.

$$cS_y(v) \equiv_{def} \forall e \in InE_{MC}(v) : \\ tsE'(e) = dQ(tsE(e), c(e)) \wedge sF_y(v)$$

Denote rTS_x as the release time of the space of tokens consumed by a firing of v.

$$rTS_x(v) = \begin{cases} v.st, & \text{if } x = 1; \\ v.et, & \text{if } x = 0 \end{cases}$$

The action to release space of consumed tokens according to release abstraction x is defined as rS_x.

$$rS_x(v) \equiv_{def} \forall e \in OutE_{MC}(v) : \\ tsE'(e) = eQ(tsE(e), p(e), rTS_x(v))$$

The action of a firing of actor v now affects not only the numbers of its incoming and outgoing edges, formalized by $fire(v)$, but also the MC edges. We formalize the firing action based on the memory abstraction xy as action $fire_{xy}$.

$$fire_{xy}(v) \equiv_{def} fire(v) \wedge rS_x(v) \wedge cS_y(v).$$

The effects of $fire_{xy}$ is exampled in Fig. 5. At state s, actor B is ready for a firing. The time stamps of tokens produced on edges of $OutE(B)$ do not affected by the abstractions. They are always $B.et$, although the values of $B.et$ may differ with respect to claim abstraction y. The time stamps of tokens on $OutE_{MC}(B)$, however, are decided by the abstraction used.

When the release time of consumed tokens is set to the end of the firing, i.e. $x = 0$, these time stamps are tagged with $B.et$, e.g. states s_{01} and s_{00}; otherwise, they are tagged with $B.st$, e.g. states s_{11} and s_{10}. Abstraction y affects the start time of a firing. When the required space is claimed at the start of the firing, i.e. $y = 1$, $B.st$ is set to be the largest time stamp of the required tokens for the firing, e.g. fire_{01} and fire_{11}. Otherwise, the largest time stamp of the required tokens for the firing can be seen as the end time of the firing, i.e. $B.et$, then $B.st = B.et - t(B)$, e.g. fire_{00} and fire_{10}.

For different abstractions, the behavior of G_{MC} is defined by $LTS(G_{MC}, xy)$ with enabling condition $readyF_{MC}$ and action fire_{xy}. Execution STE_{xy} can be obtained by a variant of Algorithm 1, in which $readyF$ and fire are replaced with $readyF_{MC}$ and fire_{xy}, respectively. $IB_{xy}(G_{MC})$ can be computed using Eqn. (1). Then we obtain the minimal achievable IP of SDFG G under memory constraint MC based on memory abstraction xy by $IB_{xy}(G_{MC})$.

Theorem 3. $IP_{xy}(G, MC) = IB_{xy}(G_{MC})$.

C. Discussions on Concurrent Events

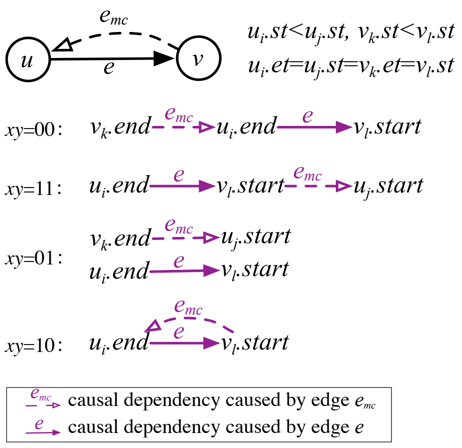
The propriety of an IP analysis is alway based on some assumptions on the corresponding scheduling of graphs. Denote $v.start$ and $v.end$ as the events of the start and the end of a firing of actor v, respectively. A schedule of an SDFG is in fact an arrangement of events on the time line. When an event occurs can be captured by $v.st$ and $v.et$ in an STE_{xy}. The difficult point is to find causal dependencies among concurrent events. When a token produced by an event is required by a concurrent event, a *causal dependency* exists between the two events.

Since we focus on a separate memory model, only events of firings of adjacent actors may be causally dependent on each other. Consider an edge $e = \langle u, v\rangle \in E$ and its MC edge e_{mc} (depicted in Fig. 6). Suppose that u_i and u_j with $i < j$ are two firings of u, and v_k and v_l with $k < l$ are two firings of v. Without lose of generality, assume $u_i.st < u_j.st$ and $v_k.st < v_l.st$. The causal dependencies may exist among simultaneous events $u_i.end$, $u_j.start$, $v_k.end$ and $v_l.start$. The following discussions are set at a time point z and $u_i.et = u_j.st = v_k.et = v_l.st = z$.

Denote uT as the set of tokens produced by $u_i.end$ on e and vT as the set of tokens required by $v_l.start$ on e. When $uT \cap vT \neq \emptyset$, the tokens produced by u_i are required by v_l. Event $v_l.start$ is causally dependent on $u_i.end$, no matter which memory abstraction is chosen.

Denote uS as the set of tokens claimed by $u_j.start$ or $u_i.end$ on e_{mc} and vS as the set of tokens released by $v_l.start$ or $v_k.end$ on e_{mc}. When $uS \cap vS \neq \emptyset$, the space claimed by u includes some of those released by v. It is caused by memory constraints. This kind of causal dependencies is concerned with memory abstractions.

The abstraction 01, which is used in [15], [7], [6], [8], is the most conservative one. The semantics of MC SDFGs under



e_{mc}
u
e
v
$u_i.st<u_j.st,\ v_k.st<v_l.st$
$u_i.et=u_j.st=v_k.et=v_l.st$
xy=00: $v_k.end \xrightarrow{e_{mc}} u_i.end \xrightarrow{e} v_l.start$
xy=11: $u_i.end \xrightarrow{e} v_l.start \xrightarrow{e_{mc}} u_j.start$
xy=01: $v_k.end \xrightarrow{e_{mc}} u_j.start$
$u_i.end \xrightarrow{e} v_l.start$
xy=10: $u_i.end \xrightarrow{e} v_l.start$, e_{mc}
causal dependency caused by edge e_{mc}
causal dependency caused by edge e

Fig. 6. The causal dependencies among concurrent events under abstraction xy.

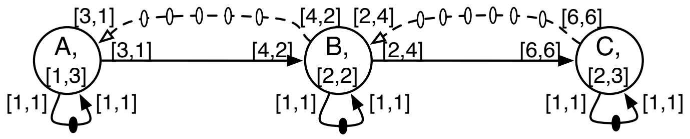


Fig. 7. MC CSDFG of G_2 under buffer bound $MC = (5, 6)$ corresponding to edges $\langle A, B\rangle$ and $\langle B, C\rangle$.

01 has no difference from semantics of 'pure' SDFGs in fact. The dependency modeled by MC edge e_{mc} is the same as that by e. Therefore $u_j.start$ is causally dependent on $v_k.end$.

Contrarily, the abstraction 10 is the most extreme one. The space on e_{mc} required by $u_i.end$ is released by $v_l.start$. Event $u_i.end$ is causally dependent on $v_l.start$. Recall that a reverse causal dependency is caused by edge e. These two events causally dependent on each other to form a cycle. That is why the IPs of some chain-structured graph in our experiments are zero.

Under the abstraction 00 (11), The space on e_{mc} required by $u_i.end$ ($u_j.start$) is released by $v_k.end$ ($v_l.start$). Therefore $u_i.end$ ($u_j.start$) is causally dependent on $v_k.end$ ($v_l.start$).

The causal dependencies among concurrent events under abstraction xy is depicted in Fig. 6.

VI. Throughput Analysis of Memory Constrained CSDFGs

The definition of memory-constraint CSDFGs (MC CSDFGs) is the same as Def. 2, providing that $c(e)$ and $p(e)$ are vectors rather than numbers. The MC CSDFG of G_2 (Fig. 2) under buffer bound $MC = (5, 6)$ corresponding to edges $\langle A, B\rangle$ and $\langle B, C\rangle$ is shown in Fig. 7. Since there are no tokens on edges $\langle A, B\rangle$ and $\langle B, C\rangle$, there are 5 and 6 tokens on their MC edges, respectively.

CSDFGs generalize SDFGs by allowing cyclicly changed sample rates and execution times for actors. Each actor has N firing pattens. The j^{th} firing of v takes $t(v)[i]$ units of time, consumes $c(e)[i]$ tokens from each $e \in InE(v)$ and produces $p(e)[i]$ tokens on each $e \in OutE(v)$, where $i = j \mod N$.

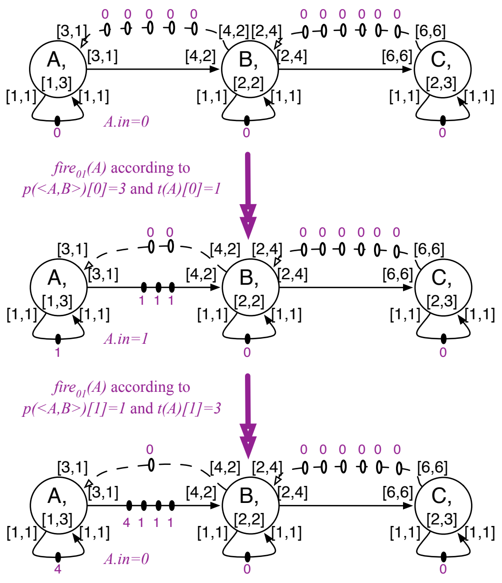

fire01(A) according to p(<A,B>)[0]=3 and t(A)[0]=1
fire01(A) according to p(<A,B>)[1]=1 and t(A)[1]=3
A.in=0
A.in=1
A.in=0

Fig. 8. The effects of firings of MC CSDFGs.

The firing also claims and released tokens on the corresponding MC edges.

The semantics of MC CSDFGs extends the semantics of MC SDFGs by taking into account the firing patterns, each of which is modeled by its index $v.in \in [0, N-1]$. Let the enabling condition and firing action defining the LTS of an MC CSDFG be $readyFC_{MC}$ and $fireC_{xy}$, respectively.

The enabling condition of a firing of actor v, $readyFC_{MC}(v)$, is a variant of $readyF_{MC}(v)$, in which all $t(v)$, $c(e)$ and $p(e)$ are replaced with $t(v)[v.in]$, $c(e)[v.in]$ and $p(e)[v.in]$, respectively. Let action $incP(v)$ increases $v.in$ by 1.

$$incP(v) \equiv_{def} v.in' = (v.in + 1) \mod N$$

The firing of actor v under abstraction xy is formalized as:

$$fireC_{xy}(v) \equiv_{def} fire_{xy}(v)[v.in] \wedge incP(v),$$

where $fire_{xy}(v)[v.in]$ is a variant of $fire_{xy}(v)$, in which all $t(v)$, $c(e)$ and $p(e)$ are replaced with $t(v)[v.in]$, $c(e)[v.in]$ and $p(e)[v.in]$, respectively.

An example illustrating the effects of firings of MC SDFGs is shown in Fig. 8. Abstraction 01 are used in the example. The first firing of actor A takes 1 unit of time to finish and produces 3 tokens on its outgoing edge $\langle A, B\rangle$ and the second firing takes 3 units of time and produces 1 token.

Execution STE_{xy} of a CSDFG can be obtained by a variant of Algorithm 1, in which $readyF$ and $fire$ are replaced with $readyFC_{MC}$ and $fireC_{xy}$, respectively. Then $IP_{xy}(G, MC)$ for CSDFG G can be similarly computed as that of an SDFG.

The $IP_{xy}(G_2, MC)$s are shown in Fig. 9. The IPs are computed under different MCs, which are storage distributions of the Pareto points obtained by the method in [6], which is under the abstraction 01.

In the original definition of CSDFGs [2], auto-concurrency is not allowed. The above defined semantics for MC CSDFGs

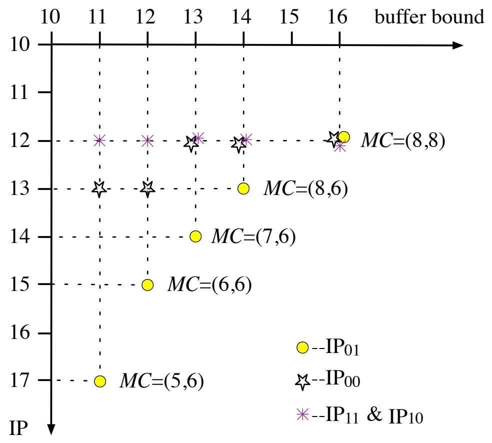


Fig. 9. $IP_{xy}(G_2, MC)$s based on abstraction xy.

is sufficient for this case. The semantics defined in [6] generalized the original definition to allow auto-concurrency. In this case, however, the overtaking problem may arise. Take actor A in G_2 for example and change its execution times to $t(A) = [3, 1]$. Actor A is enabled for two firings at the beginning. The first firing produces 3 tokens on time point 3, and the second firing produces one token on time point 1. The token produced by the second firing overtakes the tokens produced by the first firing. Formally, the monotonic property of each queue $tsE(e)$ is violated by the overtaking problem.

To deal with the problem, we change the above defined semantics on $pT(v)$ and $rTS_x(v)$. In the definitions of these two actions, $v.et$ is replaced with $\max\{v.et, preT\}$, where $preT$ is the largest one of time stamps of the last tokens on all $e \in OutE(v) \cup OutE_{MC}(v)$. This modification guarantees to preserve the monotonic property of each $tsE(e)$.

VII. Experimental Evaluation

A. Experimental Setup

We implemented our methods, variants of Algorithm 1, in SDF3 [29]. The method based on abstraction xy is named IP_{xy}. The improvements of IP_{xy} ($xy \neq 01$) compared with IP_{01} are evaluated and the execution time of IP_{xy} are shown. The MC on each graph is the smallest buffer bound that guarantees a deadlock-free execution of an SDFG. The buffer bound is the storage distribution of the first Pareto point obtained by the method in [6]. We performed experiments on two sets of SDFGs, running on a 2.67GHz CPU with 12MB cache. The experimental results are shown in Tables I, II and III. All execution times are measured in milliseconds (ms).

The first set of SDFGs consists of five practical DSP applications, including a channel equalizer (CEer) [5], a maximum entropy spectrum analyzer (MaxES), an Mp3 playback application (Mp3) [3], a sample rate converter (SaRate) [11] and a satellite receiver (Satellite) [4].

The second set of tested models consists of 360 synthetic SDFGs generated by SDF3. To evaluate the impacts of different structures on our methods, we construct one half of them as strongly connected graphs and another half as chain-structured graphs.

The number of actors in an SDFG, denoted by $|V|$, and the sum of the elements in the repetition vector, denoted by nQ, have significant impact on the performance of the various methods. We distinguish three different ranges of $|V|$: 10-15, 20-25, and 50-65; and two different ranges of nQ: 1000-1500, and 4000-6000. Then we generate SDFGs according to different combinations of structure, $|V|$ and nQ to form 12 groups. Each group includes 30 SDFGs. The explicit difference in $|V|$ and nQ among these groups is helpful for showing how the performance of our methods changes with them.

For both sets, we consider each SDFG in two cases: with and without auto-concurrency. To analyze an SDFG without auto-concurrency, we add a self-loop with one initial token to each actor of each graph.

B. Experimental Results

Table I gives the information about and results for the practical DSP examples. There are three parts in Table I. The first part is the information on the graphs, including the number of actors ($|V|$), the number of edges ($|E|$), the sum of the elements in the repetition vector (nQ), and the sum of the elements in the memory constraint (MC); the second part shows the minimal achievable iteration period (IP_{xy}) of each graph when auto-concurrency is allowed; the third part shows the same items for the cases without auto-concurrency. The improvement of each case (other than abstraction 01) compared with IP_{01} is also shown. The executions for the practical DSP examples are fast, so we do not show their execution times.

TABLE I. EXPERIMENTAL RESULTS FOR PRACTICAL DSP EXAMPLES

Graph Information					
name	CEer	MaxES	Mp3	SaRate	Satellite
$\|V\|$	22	13	4	6	22
$\|E\|$	42	13	4	5	26
nQ	42	1288	10601	612	4515
MC	73	1065	1980	32	1542
With Auto-concurrency (IP/imp. compare with IP_{01})					
IP_{01}	47128	9216	352584	1029	576
IP_{00}	47128/0%	8192/11%	117914/67%	392/62%	336/42%
IP_{11}	47128/0%	8192/11%	236160/33%	735/29%	336/42%
IP_{10}	47128/0%	683.7/93%	116424/67%	0/100%	96/83%
Without Auto-concurrency (IP/imp. compare with IP_{01})					
IP_{01}	47128	9216	352584	1088	1320
IP_{00}	47128/0%	8447/8%	120000/66%	960/12%	1056/20%
IP_{11}	47128/0%	8192/11%	236160/33%	960/12%	1056/20%
IP_{10}	47128/0%	8192/11%	120000/66%	960/12%	1056/20%

Most cases finished in several milliseconds; the largest execution time is 30ms.

In the five models, the structure of CEer are the most complex, many data dependencies existing among its actors; SaRate is a chain-structured graph. Each IP_{xy} of SaRate has a bigger improvement than that of CEer, which in fact has no improvement. The data dependencies in Mp3 and Satellite are also relatively simple. The results of the practical DSP examples show that IP_{xy}s of models with simpler structures improve their IP_{01}s more than that of models with complex structures. Notice that the IP_{10} for SaRate is zero in case with auto-concurrency. The reason is that cyclic casual dependencies exist in its STE_{10} as we explained in Section V-C.

TABLE II. THE THROUGHPUT IMPROVEMENT COMPARED WITH IP_{01} FOR SYNTHETIC EXAMPLES (AVG/MAX)

	IP_{00}	IP_{11}	IP_{10}
scsl	0.1%/2.9%	0.1%/4.9%	0.1%/4.9%
sc	11.9%/62.1%	12.3%/61.9%	22.7%/98.6%
chainsl	0.3%/9.8%	0.3%/8.2%	0.3%/9.8%
chain	47.0%/79.2%	46.8%/71.6%	91.8%/100.0%

Tables II and III give the improvement and execution times for the synthetic examples, respectively.

The throughput improvement of different abstractions other than 01 compared with IP_{01} is shown in Table II. The experimental results show that the sizes of graphs, i.e. the values of $|V|$ and nQ, have not obvious effects on the results, we distinguish the results in Table II only according to different structures. Each point includes the average and the maximal value (AVG/MAX) of 180 cases. The improvement is small for the case without auto-concurrency (scsl and chainsl). For the cases that auto-concurrency is allowed, the improvement of chain-structured graphs (chain) is much better than that of strongly connected graphs (sc).

Table III shows execution times of IP computation using method in [6] and that using the proposed methods. For different abstractions, the execution times have no essential difference, therefore we only show that of IP_{00}. Each point includes the average and the maximal value (AVG/MAX) of 30 cases. When the tested graphs scale up, our methods are generally slower than the method in [6], because we need to record time stamps of all tokens while the method in [6] need only to record the numbers of tokens, in the STE. Nevertheless, the proposed methods are still quite fast. The slowest execution of all the cases takes only about 10 seconds.

VIII. CONCLUSIONS

In this paper, we have presented a unified framework for throughput analysis of memory constrained (C)SDFGs based on difference abstractions that decide when the space of consumed data is released and when the required space is claimed. The flexible analysis framework is helpful for implementation according to schedules based on different abstractions.

The proposed technologies can be embedded into other procedures like throughput-buffering trade-off exploration [6], optimal scheduling [8], or analyzing multi-constraint SDFG [30]. They are also possible used to analysis extensions of SDFGs like scenario-aware data flow graphs [31]. The local buffering analysis is also important for memory analysis of a system. It can be complementary to the proposed methods. Further investigation will be conducted in the future.

ACKNOWLEDGMENT

The author would like to thank the anonymous referees for their valuable comments and helpful suggestions.

TABLE III. EXECUTION TIME EVALUATION FOR SYNTHETIC EXAMPLES (AVG/MAX) (MS)

	10-15	20-25	50-65	$\lvert V\rvert$ / nQ
Strongly Connected Graphs without Auto-concurrency				
[6]	11/16	31/49	209/291	1k-1.5k*
IP_{00}	13/33	14/32	16/49	
[6]	36/60	127/436	1012/1735	4k-6k
IP_{00}	153/561	252/601	260/1354	
Strongly Connected Graphs with Auto-concurrency				
[6]	1/3	2/3	7/15	1k-1.5k
IP_{00}	14/32	15/35	19/77	
[6]	4/15	5/10	13/57	4k-6k
IP_{00}	162/932	270/683	289/1423	
Chain Structured Graphs without Auto-concurrency				
[6]	19/145	54/795	615/3748	1k-1.5k
IP_{00}	34/309	67/803	189/971	
[6]	38/69	158/825	1842/10246	4k-6k
IP_{00}	159/1089	320/1011	1042/4098	
Chain Structured Graphs with Auto-concurrency				
[6]	2/5	3/8	29/55	1k-1.5k
IP_{00}	19/62	42/116	98/381	
[6]	4/7	7/22	43/146	4k-6k
IP_{00}	146/1049	260/1345	851/3696	

* 1k=1000.

REFERENCES

[1] E. Lee and D. Messerschmitt, “Static scheduling of synchronous data flow programs for digital signal processing,” *IEEE Trans. on Comput*, vol. 36, no. 1, pp. 24–35, 1987.

[2] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, “Cyclo-static dataflow,” *IEEE Trans. on signal processing*, vol. 44, no. 2, pp. 397–408, 1996.

[3] M. Wiggers, M. Bekooij, and G. Smit, “Efficient computation of buffer capacities for cyclo-static dataflow graphs,” in *Proc. of 44th Design Automation Conf. (DAC).*, 2007, pp. 658–663.

[4] S. Ritz, M. Willems, and H. Meyr, “Scheduling for optimum data memory compaction in block diagram oriented software synthesis,” in *Proc. Acoustics, Speech, and Signal Processing Conf.*, 1995, pp. 2651–2654.

[5] A. Moonen, M. Bekooij, R. van den Berg, and J. van Meerbergen, “Practical and accurate throughput analysis with the cyclo static dataflow model,” in *Proc. of 15th Int. Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2007, pp. 238–245.

[6] S. Stuijk, M. Geilen, and T. Basten, “Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs,” *IEEE Trans. on Computers*, vol. 57, no. 10, pp. 1331–1345, 2008.

[7] O. Moreira, T. Basten, M. Geilen, and S. Stuijk, “Buffer sizing for rate-optimal single-rate data-flow scheduling revisited,” *IEEE Trans. on Computers*, vol. 59, no. 2, pp. 188–201, 2010.

[8] X. Y. Zhu, M. Geilen, T. Basten, and S. Stuijk, “Memory-constrained static rate-optimal scheduling of synchronous dataflow graphs via retiming,” in *Proc. of 17th Design, Automation and Test in Europe (DATE)*, 2014, pp. 1–6.

[9] A. Ghamarian, M. Geilen, S. Stuijk, T. Basten, A. Moonen, M. Bekooij, B. Theelen, and M. Mousavi, “Throughput analysis of synchronous data flow graphs,” in *Proc. of International Conf. on Application of Concurrency to System Design (ACSD).*, 2006, pp. 25–36.

[10] S. Bhattacharyya, P. Murthy, and E. Lee, *Software synthesis from dataflow graphs*. Springer, 1996, vol. 360.

[11] P. K. Murthy, S. S. Bhattacharyya, and E. A. Lee, “Joint minimization of code and data for synchronous dataflow programs,” *Formal Methods in System Design*, vol. 11, no. 1, pp. 41–70, 1997.

[12] W.Sung and S. Ha, “Memory efficient software synthesis with mixed coding style from dataflow graphs,” *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 8, no. 5, pp. 522–526, Oct 2000.

[13] P. Murthy and S. Bhattacharyya, “Shared memory implementations of synchronous dataflow specifications,” in *Proc. of the Conf. on Design, Automation and Test in Europe (DATE)*, 2000, pp. 404–410.

[14] P. K. Murthy and S. S. Bhattacharyya, “Buffer merging - a powerful technique for reducing memory requirements of synchronous dataflow specifications,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 9, no. 2, pp. 212–237, Apr. 2004.

[15] M. Adé, R. Lauwereins, and J. Peperstraete, “Data memory minimisation for synchronous data flow graphs emulated on DSP-FPGA targets,” in *Proc. of 34th Design Automation Conf. (DAC)*, 1997, pp. 64–69.

[16] M. Geilen, T. Basten, and S. Stuijk, “Minimising buffer requirements of synchronous dataflow graphs with model checking,” in *Proc. of the 42nd Design Automation Conf. (DAC)*, 2005.

[17] W. Liu, Z. Gu, J. Xu, Y. Wang, and M. Yuan, “An efficient technique for analysis of minimal buffer requirements of synchronous dataflow graphs with model checking,” in *Proc. of the 7th International Conf. on Hardware/Software Codesign and System Synthesis*, ser. CODES+ISSS, 2009, pp. 61–70.

[18] R. Govindarajan, G. Gao, and P. Desai, “Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks,” *The Journal of VLSI Signal Processing*, vol. 31, no. 3, pp. 207–229, 2002.

[19] A. Bouakaz, P. Fradet, and A. Girault, “Symbolic analyses of dataflow graphs,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 22, no. 2, pp. 39:1–39:25, Jan. 2017.

[20] M. Benazouz and A. Munier-Kordon, “Cyclo-static dataflow phases scheduling optimization for buffer sizes minimization,” in *Proc. of the 16th International Workshop on Software and Compilers for Embedded Systems*. ACM, 2013, pp. 3–12.

[21] Q. Ning and G. Gao, “A novel framework of register allocation for software pipelining,” in *Proc. of the 20th symposium on Principles of programming languages*, 1993, pp. 29–42.

[22] Y. Yang, M. Geilen, T. Basten, S. Stuijk, and H. Corporaal, “Automated bottleneck-driven design-space exploration of media processing systems,” in *Proc. of 13th the Conf. on Design, Automation and Test in Europe (DATE)*, 2010, pp. 1041–1046.

[23] A. Singh, A. Kumar, and T. Srikanthan, “A hybrid strategy for mapping multiple throughput-constrained applications on mpsocs,” in *Proc. 14th International Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2011, pp. 175–184.

[24] S. Stuijk, T. Basten, M. Geilen, and H. Corporaal, “Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs,” in *Proc. of 44th Design Automation Conf. (DAC).*, 2007, pp. 777–782.

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

[26] F. Baccelli, G. Cohen, G. Olsder, and J. Quadrat, *Synchronization and linearity*. Wiley New York, 1992.

[27] Y. L. Gu, X. Y. Zhu, G. Zhang, and Y. He, “Pareto optimal scheduling for synchronous data flow graphs on heterogeneous multiprocessor,” in *Proc. of 21st International Conference on Engineering of Complex Computer Systems (ICECCS)*, Nov 2016, pp. 91–100.

[28] S. Sriram and S. Bhattacharyya, *Embedded multiprocessors: scheduling and synchronization*. CRC Press, 2009.

[29] S. Stuijk, M. Geilen, and T. Basten, “SDF3: SDF For Free,” in *Proc. of 6th Int. Conf. on Application of Concurrency to System Design (ACSD)*, 2006, pp. 276–278. http://www.es.ele.tue.nl/sdf3/.

[30] X. Y. Zhu, M. Geilen, T. Basten, and S. Stuijk, “Multiconstraint static scheduling of synchronous dataflow graphs via retiming and unfolding,” *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 6, pp. 905–918, June 2016.

[31] B. Theelen, M. Geilen, T. Basten, J. Voeten, S. Gheorghita, and S. Stuijk, “A scenario-aware data flow model for combined long-run average and worst-case performance analysis,” in *Proc. of 4th International Conf. on Formal Methods and Models for Co-Design (MEMOCODE)*, 2006, pp. 185–194.