Optimizing Performance Analysis for Synchronous Dataflow Graphs with Shared Resources

Daniel Thiele, Rolf Ernst
Institute of Computer and Network Engineering
Technische Universität Braunschweig, Germany
{thiele,ernst}@ida.ing.tu-bs.de

***Abstract*—Contemporary embedded systems, which process streaming data such as signal, audio, or video data, are an increasingly important part of our lives. Shared resources (e.g. memories) help to reduce the chip area and power consumption of these systems, saving costs in high volume consumer products. Resource sharing, however, introduces new timing interdependencies between system components, which must be analyzed to verify that the initial timing requirements of the application domain are still met. Graphs with synchronous dataflow (SDF) semantics are frequently used to model these systems.**

In this paper, we present a method to integrate resource sharing into SDF graphs. Using these graphs and a throughput constraint, we will derive deadlines for resource accesses and the amount of memory required for an implementation. Then we derive the resource load directly from the SDF description, and perform a formal schedulability analysis to check if the original timing constraints are still met. Finally, we perform an evaluation of our approach using an image processing application and present our results.

I. Introduction

Streaming applications such as audio and image processing algorithms are often implemented on distributed embedded systems and come with strict latency and throughput constraints. Systems such as advanced driver assistance systems or handheld devices additionally require adherence to strict power consumption and chip area constraints demanded by their respective application domains. In contrast to these requirements, complex and high-performance applications usually implement each processing step of an algorithm as a dedicated hardware unit, each consuming valuable resources. One solution to save chip area and power is a shared resource approach for less frequently used or larger hardware units (e.g. memories, arithmetic units, or runtime-configurable components), which can be shared either within the same application or between different applications. As access to these resources must be arbitrated, their introduction into the application has a significant impact on the application's real-time aspects.

In the targeted domain, synchronous dataflow (SDF) graphs [1] are often used as a model of computation, as they explicitly expose the concurrency inherent in such applications. Here, data processing units called actors are connected via communication edges with FIFO semantics modeling data dependencies. Data communication is abstracted by tokens representing the availability of data, which actors produce and consume with certain rates. Due to their good analytical properties, SDF graphs are suited to derive performance metrics such as latency and throughput. Back-pressure effects (i.e. buffering data along a pipeline including pipeline stalls if needed) can be integrated into SDF graphs by assigning buffer sizes to each edge.

Using SDF graphs, this paper presents a method to integrate shared resources into streaming applications together with a formal analysis of the resulting application performance for a given throughput constraint. For shared resource analysis, we rely on formal methods introduced by [2] and event models [3] abstractly describing the load on these resources. With these methods, the system behavior is abstracted to its timing behavior, which is then utilized to derive worst-case response times that are used to verify timing constraints. First, we identify shared resources and their connections to specific actors. Then we derive deadlines for each actor accessing a shared resource from the required throughput, assuming periodic actor executions. These deadlines specify how long an actor execution can be maximally extended while accessing a shared resource, which introduces a certain dynamic during the execution of the SDF graph. Then we compute sufficient buffer sizes for each edge in the SDF graph to support this dynamic behavior and still meet the throughput constraint. After that, event models are derived from the SDF graph (now containing buffer sizes) for each actor accessing a shared resource using self-timed execution. These event models, which are derived from the real application behavior, are then passed into a formal performance analysis to check if the initial throughput constraint that we used to derive deadlines and buffer sizes is still met. This method enables designers to efficiently use SDF graphs to model dataflow applications containing shared resources and verify their timing constraints.

The rest of this paper is organized as follows. We discuss related work in Section II, followed by an introduction of the system model for the analysis in Section III. We then present our analysis method in Section IV and evaluate it using an example in Section V. Finally, Section VI concludes the paper.

II. Related Work

Different dataflow models can be used to describe stream processing applications. The selected SDF [1] model of computation is characterized by a high design-time and scheduling predictability. In particular, we need to be able to check our application for deadlock freedom, and also need to be able to produce artificial deadlocks during event model derivation. Rate consistency is needed to derive deadlines and bounds on token production and consumption for buffer size computation. Like in [4], we annotate actors with execution times for performance analysis.

In contrast to SDF, the boolean dataflow (BDF) [5] model allows actors to produce and consume tokens dependent on a two-valued function. This is accomplished by introducing control tokens, making BDF turing complete. This implies that reasoning about deadlock behavior becomes undecidable and rate consistency cannot be guaranteed. Kahn process networks [6] are also more expressive than SDF graphs, but have the drawback that, in general, a decision of the boundedness of buffers on edges cannot be made. In [7], an approach to integrate homogeneous SDF (HSDF) graphs (i.e. graphs with token production and consumption rates equaling one) into a compositional performance analysis framework is presented. The authors describe methods for deriving performance metrics (i.e latency and throughput) and event models from HSDF graphs. SDF graphs can be converted into HSDF graphs [4], so that these methods can be applied as well. During this conversion, actors and channels are duplicated depending on their production and consumption rates and tokens are redistributed to mimic the initial state of the SDF graph. This significantly increases the complexity of the graph to be analyzed and hinders performance analysis of duplicated actors, as multiple actors must be tracked and the results must be merged. In order to cover this issue, we will present methods that are able to derive these performance metrics directly from SDF graphs. Furthermore, the impact of shared resources within the SDF graph and the derivation of deadlines under a throughput constraint are not considered in [7].

An approach to calculate buffer sizes for SDF graphs given a throughput constraint is presented in [8]. Linear bounds on token production and consumption times are derived for each edge in the graph based on strictly periodic schedules for each actor. We will extend this method to also work under the presence of the corner cases that can occur when we allow actors to prolong their execution up to their deadlines. There are also other buffer sizing methods available [9], which can be extended to account for a throughput constraint. However, the abstraction in [8] of representing the communication between actors by the accumulated number of tokens at a given time is well-suited for our extension that considers the worst-case scenario that can occur while two actors communicate.

In [10], a method to integrate resource sharing into SDF graphs based on Latency-Rate ($\mathcal{LR}$) servers is described. Arbiters in the $\mathcal{LR}$ class serve all requests at a guaranteed rate with a bounded maximum latency. The authors demonstrate that $\mathcal{LR}$-based arbiters can be realized using two SDF actors modeling latency and rate. While many arbitration schemes fall in the $\mathcal{LR}$ class (e.g. round-robin), priority-based methods, in general, do not (with the exception of rate-controlled priority arbiters). We will present a method that is not restricted to $\mathcal{LR}$ servers, but also allows priority-based arbitration in SDF analysis. This is useful when a latency critical part of an application shares a resource with latency tolerant parts.

III. System Model

For this paper, we assume a dataflow architecture comprising individually (point to point) interconnected data processing units, a set of shared resources, and a constraint on throughput. All data processing units are implemented as individual hardware units. Thus neither inteconnect nor shared resources, except for those we explicitly model, can cause contention.

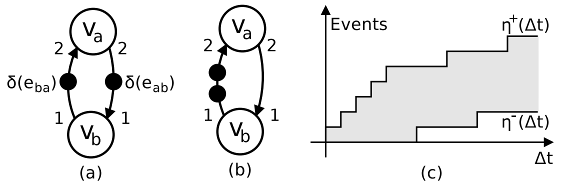

Fig. 1. Example of an SDF graph (a) and (b), and event model bounds (c)

We model the dataflow architecture using an SDF graph $G{=}(V,E,\delta,\rho,\pi,\gamma)$. Here, actors $v_i{\in}V$ model data processing units, and directed edges $e_{ij}{\in}E$ describe causal data dependencies between two actors v_i and v_j. A token distribution $\delta{:}E{\rightarrow}\mathbb{N}$ specifies the initial number of tokens available on each edge. Edges are assumed to have FIFO behavior. Let $\gamma{:}E{\rightarrow}\mathbb{N}^*$ ($\pi{:}E{\rightarrow}\mathbb{N}^*$) describe the number of tokens consumed (produced) from an edge when an actor adjacent to that edge begins (completes) its execution. The execution time of an actor is given by $\rho{:}V{\rightarrow}\mathbb{N}^*$. An actor v_i is enabled if at least $\gamma(e_{ji})$ tokens are available on all of its input edges. After an actor is activated, it consumes $\gamma(e_{ji})$ tokens from each input edge and, after an execution time of $\rho(v_i)$ clock cycles, produces $\pi(e_{ik})$ tokens on all output edges. This process is called firing. For HSDF graphs $\pi(e_{ij}){=}\gamma(e_{ij}){=}1$ holds for all e_{ij}. An SDF graph is said to be (rate) consistent if it produces on each edge as many token as it consumes from that edge in the long term run. This means the graph is able to execute infinitely often with finite memory [1]. If the initial token placement allows the graph to actually execute infinitely often, the graph is said to be deadlock-free [1]. We do not allow concurrent execution of the same data processing unit, and assume that each actor is equipped with an implicit self-edge with one initial token preventing overlapping executions.

If each actor starts firing as soon as it is enabled, the graph executes self-timed. Self-timed SDF graphs show monotonic behavior, which means that a decrease in any actor start time cannot lead to an increase of any other start time [8]. After a transient phase of length τ, consistent deadlock-free SDF graphs enter a periodic phase of length μ during which each actor v_i is executed $N\cdot q(v_i)$ times with $q{:}V{\rightarrow}\mathbb{N}$. Where N can be determined from the number of tokens on the graph's critical cycle [4] and q specifies the number of times each actor must fire to return the graph into its initial state.

The original SDF model assumes unlimited edge capacities. To model systems with limited resources, a bound on the buffer size between two actors v_i and v_j is modeled by introducing an edge e_{ji} with $\delta(e_{ji})$ representing the free FIFO buffers between v_i and v_j, whereas $\delta(e_{ij})$ describes the buffers that are currently in use. An example is shown in Fig. 1a.

A path from actor v_1 to v_n is defined to be a non-empty sequence $Q(v_1,v_n){=}(v_1,v_2,...,v_n)$, with edges $e_{ij}{\in}E$, $1{\leq}i{<}n, 1{<}j{\leq}n$ connecting these actors. If there exists a path $Q(v_i,v_i)$, then the graph is said to contain a cycle. For the scope of this paper, we only consider applications modeled by SDF graphs that do not contain cycles apart from the previously mentioned self-edges and the back edges modeling the buffer capacity between two actors. This limitation does not impose a problem for the major parts of stream processing applications, which often do not contain cycles in their data flow (e.g. optical flow, stereo vision, and marker classification).

Actors may access shared resources during execution. Let the shared resources be $s_i \in S$. The mapping $\sigma: V \rightarrow S$ associates actors with shared resources, and $\tilde{\rho}: V \rightarrow \mathbb{N}^*$ specifies the time an actor needs to access the resource without any interference from other actors. If v_i does not access a shared resource, then $\tilde{\rho}(v_i) = 0$. Each actor is allowed to access only one shared resource. This imposes no limitation in practice, as actors accessing multiple shared resources can be split into multiple actors accessing only one resource. As actors always consume tokens before they access a shared resource, such an access will lead to a prolonged actor execution time.

Shared resources can be arbitrated using different methods. Accesses to shared resources are assumed to be independent of each other. This might not be the case in practice, where actors are often causally dependent on each other. However, this simplification can only lead to overestimations of response times, as benefits by actor correlations are left unexploited.

Whenever an actor accesses a shared resource, it sends an event to the resource (e.g. memory access, arbitration request). Using an SDF description, a derivation of event models is possible [3]. These models are described by $\eta^+_{v_i}(\Delta t)$ and $\eta^-_{v_i}(\Delta t)$, the maximum and minimum number of events that actor v_i may send in a time interval Δt (see Fig. 1c). The parameters P, J, and d^{min} are used in the following to define these models such that period, jitter, and the minimum distance between any two events are specified, respectively.

$$\eta^+_{v_i}(\Delta t) = \min\left(\left\lceil \frac{\Delta t}{d_i^{min}} \right\rceil, \left\lceil \frac{\Delta t + J_i}{P_i} \right\rceil\right) \tag{1}$$

$$\eta^-_{v_i}(\Delta t) = \max\left(0, \left\lfloor \frac{\Delta t - J_i}{P_i} \right\rfloor\right) \tag{2}$$

A throughput constraint is given by the tuple (v_t, P_t) and specifies the constraining actor v_t together with a period $P_t \in \mathbb{N}^*$. The throughput of v_t is thus $\frac{1}{P_t}$. Throughout this paper, v_t denotes the throughput constrained actor. We assume v_t to execute strictly periodically, which does not impose a limitation for streaming applications, as typical input sources such as cameras or memories exhibit this execution pattern.

IV. Schedulability Analysis

The schedulability analysis comprises four steps, each of which described in detail in the following subsections. During the computation of deadlines and buffer sizes, we execute the SDF graph under certain worst-case schedules. These schedules, however, are only auxiliary means and do not impose a strict schedule on the application, which runs entirely event-triggered. Thus, we must assume self-timed execution for event model derivation and shared resource analysis.

A. *Deriving Deadlines from Throughput Constraints*

First, deadlines D_i for all actors accessing shared resources will be derived. Given an SDF graph and a throughput constraint (v_t, P_t), the period of each actor v_i can be calculated as $P_i = \frac{q(v_t)}{q(v_i)} P_t$. If every actor executes according to its period, then the throughput constraint will be met. Thus, we assume $D_i = P_i$ for actors accessing shared resources.

The buffer size on the edge between every pair of actors must be large enough to support this behavior. Assume, for example, in Fig. 1b $\rho(v_a) = \rho(v_b) = 2$, $\delta(e_{ab}) = 0$, $\delta(e_{ba}) = 2$,

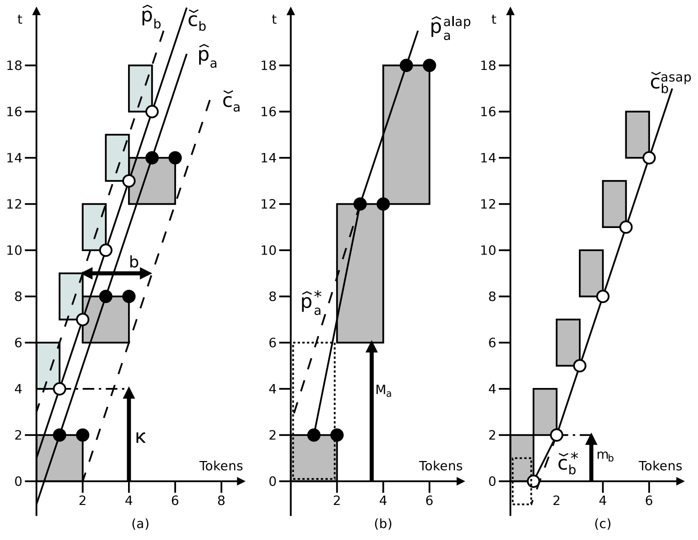

Fig. 2. Buffer size analysis on edge e_{ab} from [8] (a) and bounds for dynamic actor execution (b) and (c)

and the constraint $(v_a, 6)$ yielding $P_b = 3$. Starting v_a at t_0, v_b can first start at $t_0 + \rho(v_a)$. If v_b executes strictly periodically it will finish its second execution at $t_0 + \rho(v_a) + P_b + \rho(v_b) = t_0 + 7$, whereas v_a needed to be activated at $t_0 + 6$ to meet its period. Thus, a buffer size of $\delta(e_{ba}) = 2$ is not sufficient. In subsection IV-B, buffer sizes will be computed accordingly.

For cycles other than the cycle introduced to model buffer sizes between two actors, $D_i = P_i$ does not hold. In general cycles, deadlines are limited by the actor dependencies in such cycles, meaning that we cannot push back deadlines arbitrarily.

B. *Buffer Size Computation*

The buffer size computation is a revised approach of [8] and provides several extensions. In [8], the buffer size for each edge e_{ab} (i.e. $\delta(e_{ba})$) is computed based on linear bounds on token production and token consumption times, assuming strict periodic schedules $\nu(v_a, e_{ab})$ for all actor firings. A linear upper bound on the production time of token $x \in \mathbb{N}^*$ on e_{ab} under $\nu(v_a, e_{ab})$, assuming v_a produces tokens as late as possible, is given by

$$\hat{p}_a(x, e_{ab}, \nu(v_a, e_{ab})) = s(v_a, 1, \nu(v_a, e_{ab})) + \rho(v_a) + \frac{P_a}{\pi(e_{ab})}(x - \delta(e_{ab}) - 1) \tag{3}$$

and for a lower bound on the consumption time of token x from e_{ab} by v_b under $\nu(v_b, e_{ab})$, assuming v_b consumes tokens as early as possible, is given by

$$\check{c}_b(x, e_{ab}, \nu(v_b, e_{ab})) = s(v_b, 1, \nu(v_b, e_{ab})) + \frac{P_b}{\gamma(e_{ab})}(x - \gamma(e_{ab})) \tag{4}$$

where $s(v_i, 1, \nu(v_i, e_{ij}))$ is the time of the first activation of v_i under$\nu(v_i, e_{ij})$. Note that for the production and consumption rates $\frac{P_a}{\pi(e_{ab})} = \frac{P_b}{\gamma(e_{ab})}$ holds in rate consistent graphs.

For these bounds, tokens on e_{ab} must be produced before they are consumed, i.e. $\hat{p}_a \leq \check{c}_b$. Given this constraint using (3) and (4), a minimum difference of the start times of the actors adjacent to e_{ab} can be derived as

$$\kappa = s(v_b, 1, \nu(v_b, e_{ab})) - s(v_a, 1, \nu(v_a, e_{ab})) \geq \frac{P_a}{\pi(e_{ab})}(\gamma(e_{ab}) - \delta(e_{ab}) - 1) + \rho(v_a). \tag{5}$$

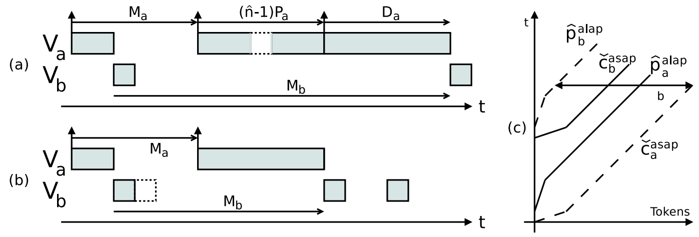

Fig. 3. Longest time between activations of v_b on e_{ab} (a) and (b) and buffer size computation for dynamic actor executions (c)

All these (per-edge) differences are then used to formulate a min-cost max-flow problem, which is solved for the entire graph to minimize the start times on all edges and the dependencies they impose on the actors. The buffer size for e_{ab} is the token backlog needed on the back edge e_{ba} (b in Fig. 2a), assuming interchanged actor roles (i.e. v_a consumes tokens as soon as possible and v_b produces tokens as late as possible). Note that these buffer sizes do not impose a strict schedule on the application's self-timed execution. They do, however, guarantee that the application cannot execute slower than we assumed during buffer size computation, as, due to monotonicity of the self-timed execution, earlier actor starts cannot lead to later actor firings. Details can be found in [8].

As we allow actors to prolong their execution up to their deadlines, we cannot assume $\nu(v_a, e_{ab})$ to be periodic anymore. This dynamic execution behavior can lead to transient shorter or longer times between firings. In the following, we will derive bounds for the maximum and minimum time between the actions of v_b on edge e_{ab} that account for this transient behavior, which will then be used to calculate bounds on token production and consumption times.

In the following, we assume $\alpha_a = D_a$ if v_a accesses a shared resource and $\alpha_a = \rho(v_a) + \tilde{\rho}(v_a)$ otherwise. Given that $D_a \leq P_a$ holds, to derive the maximum difference M_b between successive firings of v_b, we must consider two cases. If multiple firings of v_a are required to activate v_b once ($\frac{q(v_a)}{q(b_a)} > 1$), the maximum difference occurs if v_a executes with $\rho(v_a) + \tilde{\rho}(v_a)$ just once (enabling v_b as soon as possible), experiences M_a, and then executes with D_a as often as required (i.e. $\lceil \frac{q(v_a)}{q(v_b)} \rceil$ times) to activate v_b again. As $D_a \leq P_a$, v_a must wait for P_a between activations, meaning it must wait for $(\hat{n}_{ab} - 1)$ periods, where $\hat{n}_{ab} = \lceil \frac{q(v_a)}{q(v_b)} \rceil$ (Fig. 3a)

$$M_b \leq (\hat{n}_{ab} - 1)P_a + M_a - \rho(v_a) - \tilde{\rho}(v_a) + \alpha_a. \tag{6}$$

If every firing of v_a enables at least one firing of v_b ($\frac{q(v_a)}{q(b_a)} \leq 1$), then the maximum difference M_b occurs if v_a executes with $\rho(v_a) + \tilde{\rho}(v_a)$ just once, experiences M_a, and then executes with D_a. Equation (6) already accounts for this, as in this case $\hat{n}_{ab} = 1$ (Fig. 3b). This overestimates M_b, as there could be a second activation of v_b enabled by v_a (dotted box in Fig. 3a).

The smallest transient difference between two firings of v_b can, in this case ($D_b \leq P_b$), be bounded to $m_b \geq \rho(v_b) + \tilde{\rho}(v_b)$.

As actors in SDF graphs are causally dependent along paths, M_i has to be propagated to all actors. Since input actor v_t executes strictly periodically, we start with $M_t = P_t$. For actors that join multiple paths with different M_i, $\max i(M_i)$ is taken, as sufficient tokens on all these paths are required to fire.

Next, we extend the method described in [8] to support this dynamic actor execution by deriving new bounds $\hat{p}_i^{alap}$ and $\check{c}_i^{asap}$. First, we derive a new upper bound on token production times $\hat{p}_i^{alap}$. We observe that actor v_t experiences M_t between the first two firings, the token production times are maximally apart. Since v_t executes strictly periodically, the next token production time can be delayed maximally by P_t (i.e. M_t can only be experienced once), meaning that no other periodic firing sequence can lead to later token production times. This holds for all other actors, as the effects of M_t are propagated along the graph. A similar argument holds for the lower bound on token consumption times $\check{c}_i^{asap}$. The first two firings move as close together as possible if the first one is delayed maximally and the second one begins right after the previous one finished. Regarding P_i, all consecutive activations of v_i must be P_i apart. So no other periodic firing sequence can consume tokens faster without violating its period.

Fig. 2b shows the new upper bound $\hat{p}_a^{alap}$ for token production times. By overestimating the upper bound to a straight line $\hat{p}_a^*$ (dashed line), it can be embedded into the analysis of [8]. This is achieved by beginning at $\hat{p}_a$, assuming deadlines as actor execution times for $\hat{p}_a^*$, and moving it up along the Y-axis until $\hat{p}_a^* \geq \hat{p}_a^{alap}$ (dotted box indicating an actor execution). In the worst-case, M_a is experienced once at the beginning, so the Y-offset can be calculated to $M_a - P_a$, yielding

$$\hat{p}_a^*(x, e_{ab}, \nu(v_a, e_{ab})) = \hat{p}_a(x, e_{ab}, \nu(v_a, e_{ab})) + M_a - P_a. \tag{7}$$

This is done analogously for the lower bound on token consumption times $\check{c}_b^{asap}$ (Fig. 2c). This bound is embedded into [8] by underestimating the lower bound to a straight line $\check{c}_b^*$ (dashed line), which is done by beginning at $\check{c}_b$ and moving it down along the Y-axis until $\check{c}_b^* \leq c_b^{asap}$. In the worst-case, m_b is experienced once at the beginning, leading to a Y-offset of $P_b - m_b$, yielding

$$\check{c}_b^*(x, e_{ab}, \nu(v_b, e_{ab})) = \check{c}_b(x, e_{ab}, \nu(v_b, e_{ab})) - P_b + m_b. \tag{8}$$

As (7) and (8) have the same slopes as (3) and (4), the graph will still meet its throughput constraints after an initial delay.

Similar to (3), (4), and (5), $\hat{p}_a^*$ and $\check{c}_b^*$ are then used to derive minimum start time differences of the actors adjacent to e_{ab}. Then [8] can be used to minimize these start times on all edges. Fig. 3c shows how to derive buffer size from the token backlog b between the new bounds on e_{ba}, assuming interchanged roles of the adjacent actors. Here, [8] can also be used if we overestimate (underestimate) the bounds for $\hat{p}_b^{alap}$ ($\check{c}_a^{asap}$) accordingly. This is similar to Fig. 2a. To avoid clutter, the boxes showing actor executions are omitted in Fig. 3c.

So far, we have assumed shared resource accesses while deriving these bounds. For actors that do not access shared resources, tighter bounds can be given. Here, $\rho(v_a)$ can be used instead of D_a for all firings of v_a after the first one in $\hat{p}_a^{alap}$, and v_b in $\check{c}_b^{asap}$ can be assumed to execute periodically.

C. Deriving Event Models from SDF Graphs

In this step, we derive event models for all actors accessing shared resources. We extend the method presented in [7] to enable a direct application on SDF graphs. In contrast to the previous steps, this also works for cyclic graphs. Note that throughout this subsection, the SDF graph is assumed to execute self-timed. This is necessary, as in this step, we derive performance metrics for a real event-triggered (i.e. as tokens arrive) application, which can execute faster than according to a given schedule as assumed during the previous steps. This can lead to (transient) high loads on a shared resource.

1) Maximum Load: We construct a token placement that maximizes the output load of a given actor v_a. This is analogous to minimizing the distances between subsequent firings of v_a. In [7], this has been done for HSDF graphs. All tokens available in the graph are placed on all edges leading towards v_a, starting with the edges adjacent to v_a. The authors show, that if the graph is executed with this token placement, $v_a$will experience its maximum load. In HSDF graphs, this token placement can be constructed by disabling v_a and then simulating the graph until deadlock [7]. If we re-enable v_a and simulate until the periodic phase is reached, we can extract an event model from the time points at which v_a fires.

We will now extend this method to work on SDF graphs without the need to convert them to HSDF. In SDF graphs, the described method can lead to different token placements in the deadlocked state depending on the graph's initial token placement. Take for example Fig. 1a with an initial placement of $\delta(e_{ab}){=}\delta(e_{ba}){=}1$. Actor v_b can fire immediately and then must wait for v_a to fire before it can fire again. Assuming an initial placement of $\delta(e_{ab}){=}2$ and $\delta(e_{ba}){=}0$, v_b can fire twice before it depends on v_a again, resulting in a higher load.

To construct the token placement leading to the maximum load of v_a, we must consider all reachable token placements as starting points. This is done by disabling v_a and simulating until deadlock. Then we save the graph state (i.e. current progress of actor executions and token placement). After this, the output load is derived by self-timed simulation. Then we load the saved state and fire v_a once before disabling it again and repeating the steps above until an already processed deadlock state is encountered. During simulation, v_t is executed strictly periodically with P_t. To minimize token production times, all actors v_i execute with $\rho(v_i) + \tilde{\rho}(v_i)$.

Given the maximum load, an event model $\hat{E}_a{=}(\hat{P}_a, \hat{J}_a, \hat{d}_a^{min})$ can be derived from the time points of the i-th activation of v_a. For the period we have $\hat{P}_a{=}\frac{\mu}{f_{\mu,v_a}}$, where f_{μ,v_a} is the number of firings of v_a during the periodic phase. The jitter is calculated from (1) to $\hat{J}_a{=}\max_{0<\Delta t\leq\tau+\mu}(\eta_{v_a}^{+}(\Delta t)P - \Delta t)$. $\hat{d}_a^{min}$ equals the minimum distance between any two firings.

2) Minimum Load: The minimum load of v_a can be derived from the maximum time span between two successive firings of v_a. This interval can be bounded conservatively by the difference between the minimum and maximum latency along all paths from v_t to v_a [7]. The latency along a path $Q(v_i, v_j)$ is defined to be the time between the start of v_i until the first causal firing of v_j. Token x depends causally on all tokens consumed by an actor that lead to the production of x.

The minimum latency along a path is simply the sum of all actor execution times along that path $L_{min}(Q(v_t, v_a)){=}\sum_{v_i\in Q(v_t,v_a)}(\rho(v_i){+}\tilde{\rho}(v_i))$. The minimum latency $L_{min}(v_t, v_a)$ from v_t to v_a is the minimum of the latencies along all paths leading from v_t to v_a. Taking into account all reachable token placements on $Q(v_t, v_a)$, and the number of tokens on the feedback cycles that rejoin $Q(v_t, v_a)$, a better lower bound for L_{min} could be derived.

The maximum latency between actors v_t and v_a is defined as the longest time it takes an input token at v_t to cause a causally dependent firing of v_a. Let the activation time of v_t on an arbitrary path $Q(v_t, v_a)$ be t_0, and the time of the first causally dependent firing of v_a be t_f, then the latency along this path will be $L_{max}(Q(v_t, v_a)){=}t_f - t_0$.

As t_0 is fixed, we will compute a conservative estimation for t_f. In [7], this is done for HSDF graphs. We will present a method that will directly work on SDF graphs. First, we determine which firing of v_a is the first causally dependent one. This is done by propagating tokens along $Q(v_t, v_a)$ and firing actors as often as possible given the tokens propagated up to each actor's input edge. Assume, for example, a chain containing actors v_w, v_x, and v_y. Let $\tilde{\delta}(e_{wx})$ be the number of tokens propagated to edge e_{wx} including the initial tokens $\delta(e_{wx})$. The number of tokens on e_{xy} after firing v_x as often as possible given $\tilde{\delta}(e_{wx})$ is

$$\tilde{\delta}(e_{xy}){=}\Big\lfloor\frac{\tilde{\delta}(e_{wx})}{\gamma(e_{wx})}\Big\rfloor\pi(e_{xy}) + \delta(e_{xy}). \tag{9}$$

This is done for all actors on $Q(v_t, v_a){=}(v_t, v_1, ..., v_n, v_a)$, starting with $\tilde{\delta}(e_{t1}){=}\delta(e_{t1})$. The number of the first causally dependent firing $n_f(Q(v_t, v_a))$ on path $Q(v_t, v_a)$ is $n_f(Q(v_t, v_a)){=}\lfloor\frac{\tilde{\delta}(e_{na})}{\gamma(e_{na})}\rfloor + 1$. Even though we only consider paths and ignore cycles during this process, all tokens on cycles that share edges or actors with $Q(v_t, v_a)$ are accounted for, making this a valid simplification. Tokens on common edges are already considered during token propagation, and tokens on the other part of a cycle will eventually become available to the actor that rejoins cycle and path. This is because in a consistent and deadlock-free SDF graph, cycles are also consistent and deadlock-free.

In general, there can be multiple paths from v_t to v_a denoted by the set $\hat{Q}$. Imagine two paths Q_1 and Q_2 leading from v_t to v_a. Let $n_f(Q_1){=}n$ and $n_f(Q_2){=}m$, with $n{<}m$. This means that the n-th activation of v_a is the first one that is causally dependent on a firing of v_t (via Q_1). Thus, we have $n_f^{min}{=}\min_{Q\in\hat{Q}}(n_f(Q))$.

Using n_f^{min}, we can derive t_f by simulating the SDF graph under self-timed execution, while keeping track of how often v_a fires. During this simulation, we let each actor that accesses a shared resource execute with its execution time set to its deadline. We also assume that the throughput-constrained input actor v_t executes strictly periodically with P_t. Monotonicity during self-timed execution ensures that shorter actor execution times, which lead to earlier token production, cannot lead to a later start time of any actor. So even if an actor finishes its shared resource access before its deadline, this cannot lead to a later t_f than computed by using deadlines as the execution times.

The maximum latency can then be derived to $L_{max}{=}t_f{-}t_0$. As an SDF graph visits multiple states during execution (with states repeating themselves in the periodic phase), we must compute L_{max} for each state and use the maximum (as done in [7] for HSDF graphs). Using (2), we can derive the jitter $\check{J}_a$ for the minimum load case to

$$\check{J}_a{=}\Delta t - P_a{=}L_{max}(v_t, v_a) - L_{min}(v_t, v_a) - P_a. \tag{10}$$

A larger jitter implies a more general event model [3], so we derive the final event model for all actors v_a accessing shared resources by merging the results from the maximum and minimum load cases to

$$E_a{=}(\hat{P}_a, \max(\hat{J}_a, \check{J}_a), \hat{d}_a^{min}). \tag{11}$$

As we assumed a strictly periodic execution of the throughput constrained actor v_t during self-timed execution for both the minimum and maximum load, the length of the periodic phase μ is the same in both cases. Since v_a, in both cases, executes $q(v_a)$ times during μ, $P_a = \hat{P}_a$ holds.

D. Shared Resource Analysis

As we cannot rely on the real application to execute according to the schedules used to derive deadlines and buffer sizes, we use $\tilde{\rho}(v_i)$, the derived event models E_i, and deadlines D_i to perform a schedulability analysis to check if the throughput constraint is still met. Using techniques from [2], we implemented a static order scheduling analysis based on [11] to derive the worst-case time R_i^{max} (including $\tilde{\rho}(v_i)$) for a shared resource access s_j under the load imposed by all E_i for s_j. This is the time by which v_i's execution can be maximally prolonged. Thus, the system is schedulable if, for all $s_j \in S$, the inequalities $\rho(v_i) + R_i^{max} \leq D_i, \forall v_i \in \{v_i | \sigma(v_i) = s_j\}$ hold.

V. Experiments

To evaluate the proposed analysis method, we consider a specific part of an image processing algorithm for marker classification. An SDF graph of the application is shown in Fig. 4a. Processing is done pixel-wise. Input pixels in HSV color space and are split up into individual color components by the $split$ actor. Then, each component $c \in \{H, S, V\}$ enters a processing pipeline, where it is subtracted from the mean of a trained probability distribution (sub_c), squared in sq_c, divided by the corresponding distribution's standard deviation in div_c, and rounded by rnd_c. The results from the pipelines are added up in add and fed into exp, an exponentiation. The application is implemented as VHDL modules targeting an FPGA.

The dividers consume a significant amount of chip area, so our goal is to turn a single divider into a shared resource s_{div} (box in Fig. 4a) that is used for all pipelines ($\sigma(div_c) = s_{div}$). The dividers are fully pipelined and support a throughput of one pixel per clock cycle. As the application does not contain any feedback loops, we can model the access to the divider pipeline by just one actor div_c experiencing the arbitration. Of course, this can also be applied to all parallel pipeline stages.

The system is able to process one pixel per clock cycle and is targeted to run at 200 MHz leading to a rate of 254.31 frames per second (FPS) given input images of 1024×768. Since we only need to reach 25 FPS to meet real-time constraints, we set the throughput constraint to $(split, 10)$.

Our analysis results in the deadlines $d_{div_c} = 10$ cycles. The total amount of buffer size required is 140 FIFO slots. The derived event models are $E_{div_c} = (10, 51, 2)$. The schedulability analysis results in worst-case response times of $R_{div_c}^{max} = 9$ cycles for all c for static order scheduling. Assuming $\rho(div_c) = \tilde{\rho}(div_c) = 1$, we have that $\rho(div_c) + R_{div_c}^{max} \leq D_{div_c}$ holds for all c. Thus, the system, saving the chip area of two dividers, is schedulable, while still meeting timing requirements.

Fig. 4b shows a second example. Here, actors hp, mp, and np share a resource. Using the constraint $(str, 3)$, the event models for these actors are $(3, 6, 2)$, $(6, 15, 2)$, and $(15, 42, 2)$, respectively. Under static order scheduling, only np with $R_{np}^{max} = 7$ meets its deadline, whereas hp and mp miss their deadlines by 4 and 1 cycles. With static priority

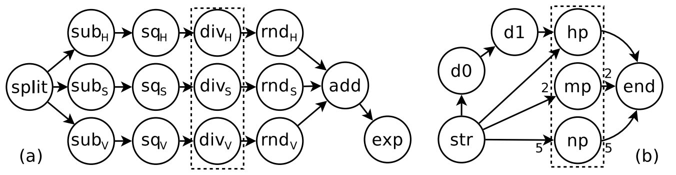

Fig. 4. Example applications

non-preemtive scheduling (higher priorities assigned to shorter deadlines), however, all actors meet their deadlines, with worst-case response times being 3, 5, and 14, respectively.

In general, response times are overestimated due to the large jitter. During schedulability analysis, the jitter causes the first shared resource accesses to be very close together, resulting in large response times due to high resource load. This is because during buffer size computation, we used worst-case assumptions and periodic schedules for each actor. The application, however, executes self-timed, leading to a dynamic execution behavior. This results in higher jitter values, especially when we derive the event model for the maximum load. There, the transient phase processing the queued up tokens increases the jitter, leading to the observed overestimation of response times.

The analysis time is in the order of seconds to minutes depending on the graph's structure and the throughput constraint.

VI. Conclusion

In this paper, we have shown how SDF graphs can be used to model dataflow applications that contain shared resources, and how to derive deadlines for accesses to these resources given a throughput constraint. We showed that this introduces a certain dynamic to the execution of the graph, and presented a method to compute buffer sizes supporting this behavior. Deriving event models from the SDF graph enabled us to integrate existing scheduling analyzes to derive performance metrics such as throughput and latency. With the help of a real-world application, we demonstrated how our approach can be used to save chip resources and still meet timing requirements.

References

[1] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, September 1987.

[2] K. W. Tindell, A. Bruns, and A. J. Wellings, "An extensible approach for analysing fixed priority hard real-time tasks," *RTSS*, 1994.

[3] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System level performance analysis - the symta/s approach," in *IEEE Proccedings Computers and Digital Techniques*. IEEE, 2005.

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

[5] J. T. Buck and E. A. Lee. (1993, Apr.) Scheduling Dynamic Dataflow Graphs With Bounded Memory Using The Token Flow Model.

[6] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," in *Information Processing '74: Proceedings of the IFIP Congress*. New York, NY: North-Holland, 1974, pp. 471–475.

[7] S. Schliecker, S. Stein, and R. Ernst, "Performance analysis of complex systems by integration of dataflow graphs and compositional performance analysis," in *DATE*, 2007, pp. 273–278.

[8] M. H. Wiggers, M. J. Bekooij, and G. J. Smit, "Buffer capacity computation for throughput constrained streaming applications with data-dependent inter-task communication," *RTAS, IEEE*, 2008.

[9] 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 *CODES+ISSS*. ACM, 2009, pp. 61–70.

[10] M. Wiggers, M. Bekooij, and G. Smit, "Modelling run-time arbitration by latency-rate servers in dataflow graphs," January 2007.

[11] R. Racu, L. Li, R. Henia, A. Hamann, and R. Ernst, "Improved response time analysis of tasks scheduled under preemptive round-robin," in *CODES+ISSS*. ACM, 2007.