Throughput-Buffering Trade-Off Exploration for Cyclo-Static and Synchronous Dataflow Graphs

Sander Stuijk, Marc Geilen, Twan Basten
Eindhoven University of Technology, Department of Electrical Engineering
{s.stuijk, m.c.w.geilen, a.a.basten}@tue.nl

***Abstract*— Multimedia applications usually have throughput constraints. An implementation must meet these constraints, while it minimizes resource usage and energy consumption. The compute intensive kernels of these applications are often specified as Cyclo-Static or Synchronous Dataflow Graphs. Communication between nodes in these graphs requires storage space which influences throughput. We present an exact technique to chart the Pareto space of throughput and storage trade-offs, which can be used to determine the minimal buffer space needed to execute a graph under a given throughput constraint. The feasibility of the exact technique is demonstrated with experiments on a set of realistic DSP and multimedia applications. To increase scalability of the approach, a fast approximation technique is developed that guarantees both throughput and a, tight, bound on the maximal overestimation of buffer requirements. The approximation technique allows to trade off worst-case overestimation versus run-time.**

***Index Terms*— cyclo-static dataflow, synchronous dataflow, buffering, throughput, optimization, Pareto analysis, trade-offs, DSP and multimedia applications**

I. Introduction

CYCLO-STATIC Dataflow Graphs (CSDFGs, [1]) and Synchronous Dataflow Graphs (SDFGs, [2]) are used to model DSP and multimedia applications [3]–[7]. The main reason for the growing popularity of these models is that they allow analysis of their timing behavior [7], [8]. This makes it possible to predict the timing behavior of an application when realized using a multiprocessor system-on-chip.

A (C)SDFG is a directed graph where the nodes (called *actors*) represent computations that communicate with each other by sending ordered streams of data-elements (called *tokens*) over their edges (called *channels*). An example of an SDFG with 3 actors a, b, c and two channels α, β is shown in Fig. 1. The execution of an actor is called a *firing*. Actor execution times are given inside actors in Fig. 1. When an actor fires, it consumes tokens from its input channels, performs a computation on these tokens and outputs the result as tokens on its output channels. An important property of an SDFG is that actors consume and produce a fixed amount of tokens on each firing. Per channel, these fixed amounts are called the consumption and production *rates*, given as edge annotations in the graph's visualization. Channels may contain initial tokens depicted as black dots annotated with their number. Storage space, *buffers*, must be allocated for the communicated tokens. The storage space influences the maximal throughput that can be achieved. In [9], a technique is presented to compute the trade-offs between the throughput and buffer size for an SDFG. These trade-offs are Pareto points in the throughput/buffer size space. An example of this trade-off space is shown in Fig. 2. Each point in the space represents a distribution of storage space over the channels α and β in the SDFG of Fig. 1 that is optimal in terms of the trade-off between storage space and throughput. To explore this trade-off space, an exact design-space exploration algorithm is presented in [9] that prunes the search space without losing any Pareto points. Finding the minimal storage requirements for a deadlock-free execution (i.e. an execution with positive, non-zero throughput) for an SDFG is already known to be NP-complete [10]. Despite of the worst-case complexity, the experimental results in [9] confirm that this algorithm can be used to explore the design space of realistic DSP and multimedia applications.

An actor in an SDFG consumes and produces a fixed amount of tokens on each firing. The CSDF model relaxes this constraint by allowing the consumption and production of tokens to vary between subsequent actor firings. It requires that the amounts of tokens consumed and produced by actor firings can be captured with a repeating finite sequence (as opposed to the constant rates of the SDF model). This makes CSDF more widely applicable in modeling dataflow applications than SDF.

This paper generalizes the techniques from [9] to Cyclo-static Dataflow Graphs. The result is the first technique to compute the complete, optimal trade-off space between the throughput and buffer size for a CSDFG. The design-space exploration algorithm uses an improved termination condition when compared to the SDF version presented in [9]. As a result, it completes the search earlier, while it is still exact. Proofs for the correctness and termination of the algorithm, omitted in [9], are provided in this paper. Experimental results are reported on a larger set of realistic applications than used in [9], including not only SDF models but also CSDF models. As the buffer minimization problem is NP-hard, the analysis can occasionally become too time consuming. For the evaluated models, the technique completes within seconds (usually milliseconds), except for one case. For an H.263 decoder, which has 3255 throughput-buffering Pareto points, the technique takes 53 minutes. To improve scalability, an approximation technique is presented that can be used to explore the design space while trading off run-time of the algorithm with quality of the end result, in terms of buffer size overestimation. We give an analytical bound on the overestimation of our heuristic. The results show that the approximation heuristic scales well. When applied to the two application models for which the exact technique has the longest run-time, it approximates the throughput-buffering trade-off space within a few milliseconds and in less than half a second respectively. The minimal buffer sizes needed

This work was supported by the Dutch Science Foundation NWO, project 612.064.206, PROMES, and the EU, project IST-004042, Betsy.

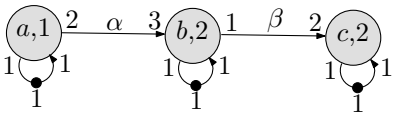


Fig. 1. Example SDFG.

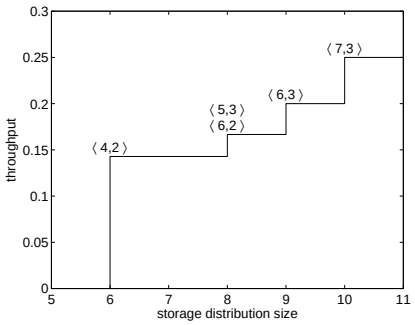


Fig. 2. Pareto space for SDFG shown in Fig. 1.

for maximal throughput are approximated with less than 0.2% and 0.5% overestimation respectively. On three artificially created models, for which the exact technique does not complete within several hours, it approximates the throughput-buffering trade-off space within seconds. The analytical bound guarantees less than 1% in overestimation in the buffer sizes needed for maximal throughput for all three models.

The remainder of this paper is organized as follows. The next section discusses related work in the area of SDFG and CSDFG buffer sizing. Sec. III formalizes the CSDF model, such that it generalizes the SDF model.[1] The operational semantics is defined in Sec. IV. The storage requirements for edges of a CSDFG are discussed in Sec. V. The throughput of a CSDFG is defined in Sec. VI. Sec. VII explains how this throughput can be computed from an execution of the graph and Sec. VIII explains how dependencies on storage space between actor firings can be identified from this execution. The dependencies are exploited in the design-space exploration algorithm presented in Sec. IX. Experimental results on the performance of the algorithm are discussed in Sec. X. Sec. XI investigates the approximation of minimal buffer capacities and Sec. XII concludes this paper.

II. Related Work

Minimization of buffer requirements in SDFGs has been studied before, see for example [3], [11]–[18]. The proposed solutions target mainly single-processor systems. Modern media applications, however, often target multi-processor systems with different approaches to scheduling and resource allocation. Furthermore, they have timing constraints expressed as *throughput* or *latency* constraints. Only looking for the minimal buffer size which gives a deadlock-free schedule as done in [3], [11]–[13], [16], [18] may result in an implementation that cannot be executed within these timing constraints. It is necessary to take the timing constraints into account while minimizing the buffers. Several approaches have been proposed for minimizing buffer requirements under a throughput constraint. In [14], a technique based on linear programming is proposed to calculate a schedule that realizes the maximal throughput while it tries to minimize buffer sizes. Hwang et al. propose a heuristic that can take resource constraints into account [15]. This method is targeted towards a-cyclic graphs and it always maximizes throughput rather than using a throughput constraint. Thus, it could lead to additional resource requirements when a lower throughput is sufficient. In [17], buffer minimization for maximal throughput of a subclass of SDFGs (homogeneous SDFGs) is studied. The proposed algorithm is based on integer linear programming. Although SDFGs can be transformed into homogeneous SDFGs. In general, the minimal buffer sizes obtained with this approach cannot be translated to exact minimal buffer sizes for arbitrary SDFGs.

A buffer minimization technique for CSDFGs is presented in [4]. The technique computes the minimal buffer requirements for a CSDFG with a static-time schedule. This schedule determines at which moment in time actor firings are started. As such, it defines the life-time of the tokens sent over the channels of the graph. The buffer requirements follow directly from the token life-times. The technique guarantees that the minimal buffer requirements are found for the given schedule, and is similar to life-time analysis techniques used in [14] and [17] for more restricted dataflow models. However, a schedule of the CSDFG may exist that realizes the same throughput with smaller buffer requirements.

In [7], a heuristic algorithm is presented that tries to minimize buffer requirements for a throughput-constrained CSDFG. The algorithm is fast but it cannot guarantee bounds on the buffer size overestimation. The reported overestimation varies between 5% and 28%.

We propose, in contrast to the existing work, an exact technique to determine all trade-offs (Pareto points) between the throughput and buffering requirements for a (C)SDFG, as well as an approximation technique to approximate this space, while providing guarantees on throughput and worst-case buffer size overestimation. An interesting observation is that both the exact and the approximation technique can also be applied after the search space has been pruned by a heuristic, which may in general lead to reduced buffering requirements for the given throughput compared to the heuristic and lower run-times compared to our exact method. For the mentioned heuristic of [7], one of our experiments shows that our technique can compute the exact result within a second when starting from the result of the heuristic. This makes our work nicely complementary to fast heuristics.

In [10], it is shown that the buffer minimization problem of homogeneous SDFGs is NP-complete. Any homogeneous SDFG is also a CSDFG and the throughput-buffer trade-off is a generalization of buffer minimization, which implies that also the buffer minimization problem for CSDFGs is NP-hard. Both our exact and our approximation technique are based on state-space exploration. Explicit state-space exploration techniques are frequently applied successfully to solve NP-complete (and sometimes worse) scheduling problems [19]–[21]. For buffer minimization, [13] proposed a state-space exploration technique to find minimal buffer requirements to execute an SDFG with a deadlock-free schedule. This motivated the investigation of explicit state-space exploration techniques in [9] and this paper. The techniques developed in [9] and in the current paper prune the search space in an efficient way, as confirmed by the experimental

[1]The original definition of a CSDFG in [1] excludes the simultaneous execution of multiple instances of the same actor, which means that according to the original definitions CSDFGs and SDFGs are not directly comparable. We present a formalization of CSDF that allows the simultaneous execution of multiple instances of the same actor.

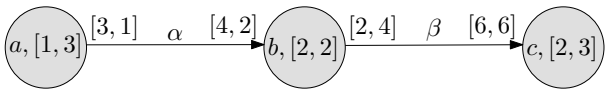


Fig. 3. Example CSDFG.

results, without losing any Pareto points.

III. Cyclo-Static Dataflow Graphs

An example of a Cyclo-Static Dataflow graph (CSDFG) is depicted in Fig. 3. Every time an actor fires it consumes a certain amount of tokens from its input ports and produces a certain amount of tokens on its output ports. These amounts are called the port *rates*. Every actor a models a periodic execution sequence $[f(0), f(1), \cdots, f(N-1)]$ of length $N \geq 1$. The meaning of this sequence is as follows. The i-th time that the actor a is fired, it executes the function $f(i \bmod N)$. As a consequence, the port rates and execution time of actors are also a sequence. These sequences are visualized as port and actor annotations (see Fig. 3). The channels in the graph may contain tokens. The storage space of a channel is in principle unbounded, i.e., it can contain arbitrarily many tokens.

Formally, a CSDFG is defined as follows. Let $\mathbb{N}$ denote the positive natural numbers, and $\mathbb{N}_0$ the natural numbers including 0. Assume a set P of ports. With each port $p \in P$, a sequence of rates $[r_0, r_1, \cdots, r_{N-1}]$ with $r_i \in \mathbb{N}_0$ and $N \in \mathbb{N}$ is associated. The number of tokens consumed or produced by a port $p \in P$ on its i-th access is given by $Rate(p, i) = r_{i \bmod N}$ (where i starts from 0).

Definition 1: (Actor) *An actor a is a tuple (I, O, T) consisting of a set $I \subseteq P$ of input ports (denoted by $In(a)$), a set $O \subseteq P$ of output ports (denoted with $Out(a)$) with $I \cap O = \emptyset$ and a sequence $T = [t_0, t_1, \cdots, t_{N-1}]$ of execution times with $t_i \in \mathbb{N}_0$.*

Definition 2: (CSDFG) *A CSDFG is a tuple (A, C) consisting of a finite set A of actors and a finite set $C \subseteq P \times P$ of channels. The channel source is an output port of some actor, the destination is an input port of some actor. All ports of all actors are connected to precisely one channel. For every actor $a = (I, O, T) \in A$, we denote the set of all channels that are connected to the ports in I (O) by $InC(a)$ ($OutC(a)$).*

In the original CSDF definition [1], no assumptions are made on the execution time of actors. The example CSDFG presented in [1] suggests that fixed actor execution times are used. In line with [4] and [7], this paper generalizes the fixed actor execution times to a sequence of execution times for each actor.

As mentioned, actor execution is defined in terms of *firings*. The execution time of the i-th firing of an actor a is denoted as $\tau(a, i)$. When actor a starts its i-th firing, it removes $Rate(q, i)$ tokens from all $(p, q) \in InC(a)$. The execution continues for $\tau(a, i)$ time units and when it ends, it produces $Rate(p, i)$ tokens on every $(p, q) \in OutC(a)$. In this paper, it is assumed that all port rate sequences and execution time sequences in a CSDFG are of the same length N. This assumption is only made for readability. The presented techniques are also valid when port rate and/or execution time sequences of different lengths are used. Every sequence can always be concatenated till its length is equal to the least common multiple of the lengths of all sequences used in the graph. Doing so for all sequences gives a CSDFG in which all sequences are of equal length.

Definition 3: (SDFG) *An SDFG is a CSDFG with the length of execution time and rate sequences N equal to one.*

For certain rates in a (C)SDFG, the (C)SDFG deadlocks or tokens accumulate on the channels. In the latter case, a (C)SDFG can only execute in unbounded memory. Consistency (CSDF [1], SDF [2]) is known to be a necessary condition to allow an execution within bounded memory in which no actors deadlock [22].

Definition 4: (Consistency, Repetition vector) *A repetition vector q of a CSDFG (A, C) is a function in $A \to \mathbb{N}_0$ given by $q(a) = N \cdot r(a)$ for all $a \in A$ where r is a function in $A \to \mathbb{N}_0$ such that for each channel $(o, i) \in C$ from actor $a \in A$ to $b \in A$, $r(a) \cdot \sum_{0 \leq k < N} Rate(o, k) = r(b) \cdot \sum_{0 \leq k < N} Rate(i, k)$. A repetition vector is called non-trivial if and only if $q(a) > 0$ for all $a \in A$. A CSDFG is called consistent if and only if it has a non-trivial repetition vector. For a consistent graph, there is a unique smallest non-trivial repetition vector which is designated as* the *repetition vector of the CSDFG.*

A repetition vector thus represents a number of firings per actor that brings the graph back to the distribution of tokens before the firings. The repetition vector of our example graph (see Fig. 3) is equal to $[6\ 4\ 2]^T$, ordering the actors from left to right. This shows that the graph is consistent. Since inconsistent graphs are typically not useful and consistency is straightforward to check [12], we restrict our attention to consistent CSDFGs. Furthermore, we assume connectedness. For unconnected graphs, analysis can be done per connected subgraph.

IV. Operational Semantics of CSDFGs

To describe and study the methods introduced in this paper, CSDFG execution is formalized through a labeled transition system. This requires appropriate notions of states and of transitions.

As explained, an actor consumes input tokens at the start of a firing, and produces output at the end of the firing. In the semantics, channels have infinite storage space, which means that there is always sufficient space available for output. Physical storage constraints are modeled by additional channels. The semantics abstracts from the actual data that is being communicated or processed by actors and treats all data elements equally in the form of tokens. This is possible as we are interested in the timing behavior and memory usage, and not for example in functional analysis. In order to capture the timed behavior of a CSDFG, we need to keep track of the distribution of tokens over the channels, of the start and end of actor firings, and the progress of time.

To measure the number of tokens present in, read from or written to channels, we define the following concept.

Definition 5: (Channel quantity) *A channel quantity on the set C of channels is a mapping $\gamma : C \to \mathbb{N}_0$. If γ_1 is a channel quantity on C_1 and γ_2 is a channel quantity on C_2 with $C_1 \subseteq C_2$, we write $\gamma_1 \preceq \gamma_2$ if and only if for every channel $\alpha \in C_1$, $\gamma_1(\alpha) \leq \gamma_2(\alpha)$. Channel quantities $\gamma_1 + \gamma_2$ and $\gamma_1 - \gamma_2$ are defined by pointwise addition resp. subtraction of γ_1 and γ_2 resp. γ_2 from γ_1; $\gamma_1 - \gamma_2$ is only defined if $\gamma_2 \preceq \gamma_1$.*

The amount of tokens read at the start of the i-th firing of some actor a can be described by a channel quantity $Rd(a, i)$ such that $Rd(a, i)(p, q) = Rate(q, i)$ if $q \in In(a)$ and $Rd(a, i)(p, q) = 0$ otherwise. Similarly, the amount of tokens produced at the end of

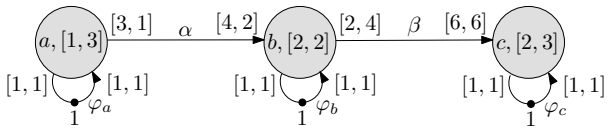


Fig. 4. Limited auto-concurrency.

the i-th firing is given by a channel quantity $Wr(a,i)$ such that $Wr(a,i)(p,q) = Rate(p,i)$ if $p \in Out(a)$ and $Rd(a,i)(p,q) = 0$ otherwise.

Definition 6: (STATE) *The state of a CSDFG (A,C) is a 3-tuple (γ, υ, η). Channel quantity γ associates with each channel the amount of tokens in that channel in that state. To keep track of time progress, an actor status $\upsilon : A \to \mathbb{N}_0^{\mathbb{N}_0 \times \mathbb{N}_0}$ associates with each actor $a \in A$ a multiset of pairs of numbers representing the remaining times of different firings of a and the index in the actor execution sequence corresponding to the firing start. $\eta : A \to \mathbb{N}_0$ associates with each actor $a \in A$ its current position in the actor execution sequence. The initial state of a CSDFG is determined by initial token distribution γ, which means that the initial state equals $(\gamma, \{(a, \{\}) \mid a \in A\}, \{(a,0) \mid a \in A\})$ (with $\{\}$ denoting the empty multiset).*

The use of a multiset of pairs of numbers to keep track of actor progress instead of a single (pair of) number(s) allows multiple simultaneous firings of the same actor (auto-concurrency). This is a generalization of the original CSDF semantics of [1], but it is in line with the standard SDF semantics (see, e.g., [23]). By allowing auto-concurrency, we achieve that our CSDF definition is a true generalization of the SDF model of computation. If desirable, auto-concurrency can always be limited or excluded by adding self-loops to actors with a number of initial tokens equivalent to the desired maximal auto-concurrency degree. For our running example, we disallow auto-concurrency by adding self-loops (channels φ_a, φ_b and φ_c) with a single token to all actors as shown in Fig. 4.

The dynamic behavior of the CSDFG is described by transitions. Three different types are distinguished: start of actor firings, end of firings, or time progress in the form of clock ticks.

Definition 7: (TRANSITION) *A transition of CSDFG (A,C) from state $(\gamma_1, \upsilon_1, \eta_1)$ to state $(\gamma_2, \upsilon_2, \eta_2)$ is denoted by $(\gamma_1, \upsilon_1, \eta_1) \xrightarrow{\beta} (\gamma_2, \upsilon_2, \eta_2)$ where label $\beta \in (A \times \{start, end\}) \cup \{clk\}$ denotes the type of transition.*

- *Label $\beta = (a, start)$ corresponds to the firing start of actor $a \in A$. This transition may occur if $Rd(a, \eta_1(a)) \preceq \gamma_1$ and results in $\gamma_2 = \gamma_1 - Rd(a, \eta_1(a))$, $\eta_2 = \eta_1[a \mapsto (\eta_1(a) + 1) \bmod N]$, i.e., η_1 with the value for a replaced by $(\eta_1(a) + 1) \bmod N$, and $\upsilon_2 = \upsilon_1[a \mapsto \upsilon_1(a) \uplus \{(\tau(a, \eta_1(a)), \eta_1(a))\}]$ (where $\uplus$ denotes multiset union).*
- *Label $\beta = (a, end)$ corresponds to the firing end of $a \in A$. This transition can occur if $(0,i) \in \upsilon_1(a)$ for some i and results in $\upsilon_2 = \upsilon_1[a \mapsto \upsilon_1(a) \backslash \{(0,i)\}]$ (where $\backslash$ denotes multiset difference), and $\gamma_2 = \gamma_1 + Wr(a,i)$, $\eta_2 = \eta_1$.*
- *Label $\beta = clk$ denotes a clock transition. It is enabled if no end transition is enabled and results in $\gamma_2 = \gamma_1$, $\eta_2 = \eta_1$, and υ_2 with for all actors $a \in A$, $\upsilon_2(a) = \{(m-1, n) \mid (m,n) \in \upsilon_1(a)\}$.*

Definition 8: (EXECUTION) *An execution of a CSDFG is an infinite alternating sequence of states and transitions $s_0 \xrightarrow{\beta_0} s_1 \xrightarrow{\beta_1} \ldots$ starting from the designated initial state s_0.*

Note that all CSDFG (even a deadlocked one, in which no actor is firing or ready to fire) has an infinite execution as time always progresses.

V. STORAGE REQUIREMENTS

As mentioned in Sec. III, channels have unbounded storage space in the semantics. However, in practice storage space must be bounded. Bounded storage space for channels can be realized in different ways. One option is to use a memory that is shared between all channels. The required storage space for the execution of a CSDFG is then determined by the maximum number of tokens stored at the same time during the execution of the graph. Murthy et al. use this assumption to schedule SDFGs with minimal storage space [16]. This is a logical choice for single-processor systems in which actors can always share the memory space. A second option is to use a separate memory for each channel, so empty space in one cannot be used for another. This assumption is logical in the context of multiprocessor systems, as memories are not always shared between all processors. The channel capacity must be determined per channel over the entire schedule, and the total amount of memory required is obtained by adding them up. Minimization of the memory space with this variant is considered in [3] and [11]. Hybrid forms of both options can be used [13]. In this paper, we assume channels cannot share memory space. This gives a conservative bound on the required memory space when the CSDFG is implemented using shared memory. In that case, the CSDFG may require less memory, but it will never require more memory than determined by our method.

The maximum number of tokens that can be stored in a channel (*channel capacity*) is captured by a *storage distribution*.

Definition 9: (STORAGE DISTRIBUTION) *A storage distribution of a CSDFG (A,C) is a channel quantity δ that associates with every $\alpha \in C$, the capacity of the channel.*

The storage space required for a storage distribution is called the *distribution size*.

Definition 10: (DISTRIBUTION SIZE) *The size of a storage distribution δ is given by: $|\delta| = \sum_{\alpha \in C} \delta(\alpha)$.*

A possible storage distribution for the CSDFG shown in Fig. 3 would be $\delta(\alpha) = 8$ and $\delta(\beta) = 6$, denoted as $\langle \alpha, \beta \rangle \mapsto \langle 8, 6 \rangle$. It has a distribution size of 14 tokens. Note that if tokens on different channels represent different amounts of data, this can easily be accounted for in the definition of distribution size. In the remainder, we assume that all tokens are of equal size.

Let (p,q) be a channel from actor a to actor b. Assume that the channel contains in the initial state of the execution d tokens. The number of tokens in the channel after n firings of a and m firings of b is given by the following equation:

$$d + \sum_{0 \le i < n} Rate(p,i) - \sum_{0 \le j < m} Rate(q,j)$$

This is equal to:

$$\left\lfloor \frac{d + \sum_{0 \le i < n} Rate(p,i) - \sum_{0 \le j < m} Rate(q,j)}{s} \right\rfloor \cdot s + d \bmod s,$$

with $s = \gcd\{Rate(p,i), Rate(q,j) \mid 0 \le i < N, 0 \le j < N\}$. The number of tokens that may ever appear in a channel, and

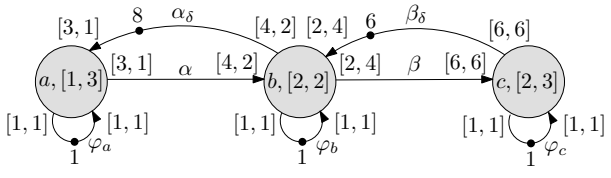


Fig. 5. CSDFG with storage distribution $\langle 8, 6\rangle$.

hence the storage space which can be effectively used, depends on the gcd of all possible combinations of rates at which the actors a and b produce and consume tokens. This gcd, s in the above formula, is called the *step size* of the channel. It follows furthermore that it is not meaningful to have a number of initial tokens d in a channel such that $d \mod s \neq 0$. These tokens will never disappear. Thus, in the remainder, we assume for readability that also the number of initial tokens in a channel is a multiple of its step size. If this assumption is not valid, computed buffer sizes can be corrected by increasing the storage space of a channel that contains d initial tokens by $d \bmod s$ tokens.

The bound on the storage space of each channel can be modeled in a CSDFG (A, C) by adding for channel $(p, q) \in C$ from an actor $a \in A$ to an actor $b \in A$ a channel (q_δ, p_δ) from b to a, where p_δ and q_δ are fresh ports not yet in use in the CSDFG, with $Rate(p_\delta, i) = Rate(p, i)$ and $Rate(q_\delta, i) = Rate(q, i)$. The number of initial tokens on the channel (q_δ, p_δ) equals the storage space of the channel (p, q) minus its own initial tokens.

In the remainder, subscript 'δ' is used to denote elements used to model storage space. The CSDFG which models the storage distribution δ in a CSDFG (A, C) is denoted (A_δ, C_δ). Fig. 5 shows the CSDFG which encodes the storage distribution $\langle 8, 6\rangle$ for our running example. Note that no storage space is allocated for the self-loops on the actors. These self-loops are introduced to model absence of auto-concurrency and will not require storage space in a real implementation and can thus be ignored. In fact, our technique allows in general to specify which channels should be considered buffers, and which channels model other dependencies. The self-loop dependencies added to limit auto-concurrency are just one example of this flexibility. The only requirement is that the CSDFG modeling the storage distribution is strongly connected, as explained below. This means that Def. 9 can be relaxed to assign channel capacities to a subset of channels only.

At the start of a firing, an actor consumes its input tokens. This includes the tokens it consumes from the channels which model the storage space of channels to which the actor will write. The consumption of these tokens can be seen as allocation of storage space for writing the results of the computation. At the end of the firing, the actor produces its output tokens. This includes the production of tokens on channels which model the storage space of channels from which the actor has read tokens at the beginning of the firing. The production of these tokens can be seen as the release of the space of the input tokens. In other words, the model assumes that space to produce output tokens is available when an actor starts firing and that space used for input tokens is released at the end of the firing. The chosen abstraction is conservative with respect to storage and throughput if in a real implementation space is claimed later, or released earlier or data tokens are read later or written earlier.

VI. THROUGHPUT

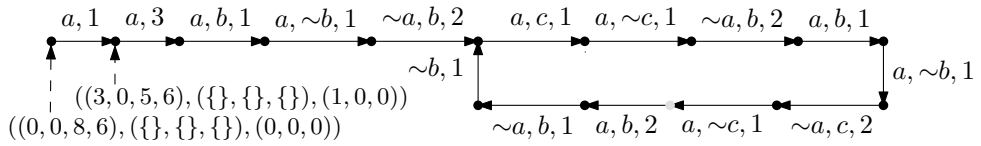
Throughput is an important design constraint for embedded multi-media systems. The throughput of a graph refers to how often an actor produces an output token. There exists a particular type of execution for SDFGs, namely self-timed execution, which gives maximal throughput [23]. In a self-timed execution, clock transitions occur only when no start transitions are enabled. It requires that each actor fires as soon as it is enabled. This execution guarantees that all actor firings occur as early as possible. So, this execution guarantees that at any moment in time the maximal number of actor firings possible has occurred since the start of the execution. Hence, the self-timed execution of an SDFG achieves maximal throughput. It is obvious that for the same reason also the self-timed execution of a CSDFG will give its maximal throughput.

Definition 11: (ACTOR THROUGHPUT) *The throughput of an actor a for the self-timed execution of a CSDFG is defined as the average number of firings of a per time unit in the execution. It is denoted with $Th(a)$.*

Note that some actors in some graphs can only achieve their maximum throughput with unbounded channels [22]. In this paper, we focus on throughput which can be achieved within bounded storage space; throughput achieved with infinite storage space cannot be implemented and is therefore not considered. A CSDFG in which all actors are connected through sequences of data dependencies and that incorporates a (by definition finite) storage distribution for all data channels is always strongly connected. In that situation, the fixed rate sequences of the actor ports ensure that the number of times actors fire with respect to each other (repetition vector) is constant. In other words, the throughput of each pair of actors in a graph is related to each other via a constant. This allows us to define a normalized notion of throughput for a CSDFG, in line with the definition for SDF given in [8]. Thus, in the remainder we assume that a CSDFG modeling a storage distribution is strongly connected.

Definition 12: (THROUGHPUT) *The throughput of a CSDFG $G = (A, C)$ is defined as $Th(G) = \frac{Th(a)}{q(a)}$, for an arbitrary $a \in A$, where q is the repetition vector of G.*

To compute the throughput of our example SDFG with the given storage distribution (see Fig. 5), we first look at the transition system of the self-timed execution as shown in Fig. 6. States are represented by dots. Sequences of state transitions consisting of all enabled start transitions, followed by a maximal number of time steps, followed by all possible end transitions (called macro-steps) are indicated by single arrows. The label with a transition indicates which actors start their firing in this transition and the elapsed time till the next depicted state is reached. Actors that continue their firings in a transition are labeled with a tilde. The transition system consists of a finite sequence of states and transitions, called the *transient phase*, followed by a sequence of states and transitions which is repeated infinitely often and is called the *periodic phase*. (The next section shows that this is always the case.) Actor c in our example is considered to determine the throughput of the example graph. This actor ends its firing for the first time after 10 clock transitions. At that moment, the graph is in the periodic phase of the schedule. The subsequent firings of c are then repeatedly executed 7 and 6 clock transitions apart. The periodic phase is repeated indefinitely. Hence, the



state vector (δ, υ, η) is encoded via a 3-tuple where δ corresponds to edges α, β, α_δ, β_δ resp., the tuple υ defines the multiset for a, b, c resp. and the tuple η defines the sequence position of a, b, c resp.; for readability, the self-edges connecting an actor to itself are omitted from the state vector.

Fig. 6. CSDF state space of the example CSDFG.

average time between firings over the whole schedule converges to the average time between firings in the periodic phase. So, the throughput of c is $Th(c) = 2/(7+6)$. Actor c fires two times according to the repetition vector of the graph. Hence, the throughput of the graph is equal to $1/13$. Essentially, this states that the self-timed execution of the graph performs one execution of the repetition vector every 13 time units.

VII. Throughput Calculation

The throughput of a CSDFG can be computed from its state space. The following result is a straightforward generalization of a similar result for SDFGs given in [8].

Theorem 1: (Periodic behavior) *The state space for any CSDFG (A, C) with storage distribution δ contains always exactly one cycle if we consider macro-steps.*

Proof: The CSDFG (A_δ, C_δ) modeling δ in (A, C) is strongly connected. This means that every actor depends on tokens from every other actor, which limits the difference between the number of firings of actors wrt each other. This implies that there exists a bound on the number of simultaneous actor firings and the number of tokens in any channel, and hence, there is only a finite number of different reachable states. Further, in the transition system, there is always at least one transition enabled (even in a deadlock state, there is still a clock transition enabled), which implies that the number of transitions that will occur is infinite. By the pigeon hole principle, at least one of the finite number of reachable states is visited infinitely often. Since the self-timed execution is deterministic (if we consider the execution in macro-steps as explained above, because simultaneous starts and ends can be arbitrarily interleaved in the semantics), there is only one transition to leave any (recurrent) state. Hence, there is exactly one cycle in the state space (in terms of macro-steps). ■

The theorem states that the state space of any CSDFG with bounded storage space for all channels consists of a transient phase followed by a periodic phase. Def. 11 defines the throughput of an actor over an execution which contains infinitely many transitions. The periodic phase is repeated indefinitely, while the states in the transient phase are visited only once. Hence, the average time between two firings over the whole execution converges to the average time between two firings in the periodic phase. So, the throughput can be computed from the periodic phase while ignoring the transient phase.

Proposition 1: (Throughput) *The throughput of an actor a in a CSDFG with some storage distribution δ is equal to the number of firings of a in one period of the periodic phase divided by the number of clock transitions in the period.*

Proof: Follows from Def. 11 and Theorem 1. ■

The throughput of a CSDFG can be computed by executing the CSDFG in a self-timed manner while remembering visited states until a state is revisited. At that point, the periodic phase is reached and the throughput of an actor can be computed using Prop. 1. The throughput of the graph can then be computed using Def. 12. The number of states that must be remembered can be kept small. We can enforce deterministic execution by choosing a fixed order among simultaneously enabled transitions in the transition system without affecting the throughput. For every actor it holds that the number of actor firings in the cycle is a multiple of its repetition vector entry [8]. Thus to detect a cycle, an arbitrary actor a (with repetition vector entry $q(a)$) can be selected, and once every $q(a)$ times the state in which an end-of-firing transition of a occurs must be stored. To detect deadlock, it must also be checked whether a clock transition remains in the same state. It is not necessary to store this state. To compute the throughput, we must additionally store the number of clock transitions between each two stored states. For our example CSDFG (see Fig. 6) and assuming that actor c with $q(c) = 2$ is selected, only the gray state must be stored.

The CSDF model of Sec. III assumes that all execution time and rate sequences are of equal length. When sequences of different lengths are used, a sequence can always be concatenated till its length is equal to the least common multiple of the lengths of all sequences used in the graph. Alternatively, the different lengths can be taken into account in the definitions of states (Def. 6) and transitions (Def. 7). The state should then store the position of an actor in the execution sequence that is the least common multiple of all sequences of that actor and the definition of start and end transitions should be adapted to use appropriate modulo operations to determine the correct execution times and correct numbers of consumed and produced tokens. It is important to note that when either the length of sequences is taken into account in the definition of states and transitions or the concatenation of sequences is used, the repetition vectors and the number of visited states in the throughput computation are identical. Hence, the complexity and the run-time of both approaches are the same.

VIII. Storage Dependencies

The maximal throughput of a CSDFG may be limited by channel capacities. In the self-timed execution of the CSDFG, an actor may, for example, be waiting for tokens on a channel α_δ (modeling the storage space of channel α). Adding tokens to α_δ (i.e. increasing the storage space of α) may enable the actor to fire earlier and possibly increase the throughput of the CSDFG. The immediate dependency of an actor firing on tokens produced by the end of another firing is called a causal dependency.

Definition 13: (Causal Dependency) *A firing of an actor a causally depends on the firing of an actor b via a channel α if*

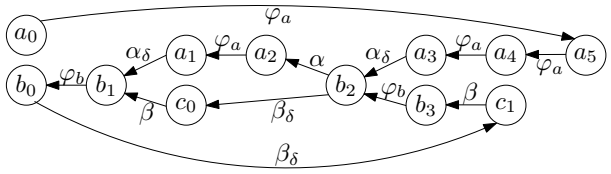


Fig. 7. Causal dependency graph of the example CSDFG.

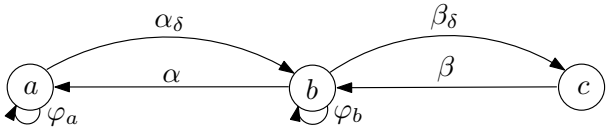


Fig. 8. Abstract causal dependency graph of the example CSDFG.

and only if the firing of a consumes a token from α produced by the firing of b on α without a clock transition between the start of the firing of a and the end of the firing of b.

If a causal dependency appears in the periodic phase of execution, the actor will repeatedly (infinitely often) not be able to fire earlier which on its turn may influence the throughput. Throughput may increase if these dependencies are resolved. All causal dependencies between the actor firings of the periodic phase can be captured in a causal dependency graph. It is sufficient if only the dependencies between actor firings in one period of the periodic phase are considered as the dependencies are equal in all periods.

Definition 14: (CAUSAL DEPENDENCY GRAPH) *Given a CSDFG (A_δ, C_δ) incorporating a storage distribution δ and a sequence p of states and transitions corresponding to a period of the self-timed execution of (A_δ, C_δ) (starting at some arbitrary state in the period). The causal dependency graph (D, E) contains a node a_k for the k-th firing in p of actor $a \in A_\delta$. The set of dependency edges E contains an edge if and only if there exists a causal dependency between the corresponding firings.*

The causal dependency graph for the CSDFG of Fig. 5 is shown in Fig. 7, assuming the gray state as the start state. The edges in the causal dependency graph represent causal dependencies between actor firings. A causal dependency goes via a set of channels in the corresponding CSDFG (see Def. 13). This association of an edge in the causal dependency graph to a set of channels in the CSDFG is left implicit in Def. 14, but it is visualized in Fig. 7 by labeling the edges in the graph. Note that the set of channels associated to a dependency edge may contain only more than one channel when the CSDFG is a multigraph.

The throughput of a CSDFG is limited by an infinite sequence of causal dependencies between the actor firings, captured by a causal dependency cycle in the causal dependency graph.

Definition 15: (CAUSAL DEPENDENCY CYCLE) *A causal dependency cycle is a simple cycle in the causal dependency graph.*

A causal dependency cycle is a sequence of actor firings that causally depend on each other, starting and ending with the same actor firing. Causal dependencies caused by channels that model storage space are of interest as adding tokens to these channels (i.e., increasing their storage space) may resolve causal dependency cycles and increase throughput.

Definition 16: (STORAGE DEPENDENCY) *Given a CSDFG (A_δ, C_δ) incorporating a storage distribution δ and its causal dependency graph Δ. A channel $\alpha \in C_\delta$ has a* storage dependency *in Δ if and only if there exists a causal dependency in some dependency cycle of Δ via channel α_δ.*

Storage dependencies can be used to determine which storage capacities can be enlarged to increase throughput of the graph. However, two issues remain to be solved. First, if the graph deadlocks, the causal dependency graph is empty and provides no information about which channel capacities to enlarge. Second, having a node for every firing of every actor, the causal dependency graph may become prohibitively large (a multiple of the sum of entries in the repetition vector). We solve the latter issue first and subsequently the deadlock case.

Cycle detection in the causal dependency graph can become very time consuming. To solve this, an abstract version of the causal dependency graph can be constructed in which the number of nodes is equal to the number of actors in the CSDFG.

Definition 17: (ABSTRACT CAUSAL DEPENDENCY GRAPH) *Given a CSDFG (A_δ, C_δ) incorporating a storage distribution δ and its causal dependency graph (D, E). The abstract causal dependency graph (D_a, E_a) contains an abstract dependency node $d_a \in D_a$ for each actor $a \in A_\delta$. For each dependency edge $(a_k, b_l) \in E$, there is an edge (d_a, d_b) in E_a.*

Fig. 8 shows the abstract causal dependency graph corresponding to the causal dependency graph of Fig. 7. As for the causal dependency graph, each dependency edge in the abstract causal dependency graph is associated with the set of channels causing this dependency. In practice, the abstract causal dependency graph of a CSDFG can be constructed by traversing through the cycle in the state space of the CSDFG once. It is not necessary to construct the underlying causal dependency graph. An important property of the abstract causal dependency graph is that it includes at least all storage dependencies present in the full causal dependency graph. (The definition of a storage dependency carries over to the abstract causal dependency graph.)

Theorem 2: (PRESERVATION OF STORAGE DEPENDENCIES) *The set of storage dependencies of an abstract causal dependency graph contains all storage dependencies of the corresponding causal dependency graph.*

Proof: Given a causal dependency graph $\Delta = (D, E)$ and its corresponding abstract causal dependency graph $\Delta_a = (D_a, E_a)$. Any dependency edge $(a_i, b_j) \in E$ from a node which corresponds to the i-th firing of actor a to the j-th firing of actor b maps in Δ_a to an edge from the abstract causal dependency node of actor a to the abstract causal dependency node of actor b. Any cycle of dependencies in Δ is a sequence of dependency edges which in the abstract causal dependency graph (using the mapping of causal dependency nodes in Δ to nodes in Δ_a) is also a sequence starting and ending in one node, i.e., it is also a cycle in Δ_a. Hence, the edge is also on some cycle in Δ_a and thus also on some simple cycle. ■

In the case that the CSDFG deadlocks (because of a shortage of storage space or an inherent deadlock due to the specified data and control dependencies), the regular causal dependency graph is empty by definition. To find out which channel capacities need to be increased, if any, the following alternative definitions are used in that case.

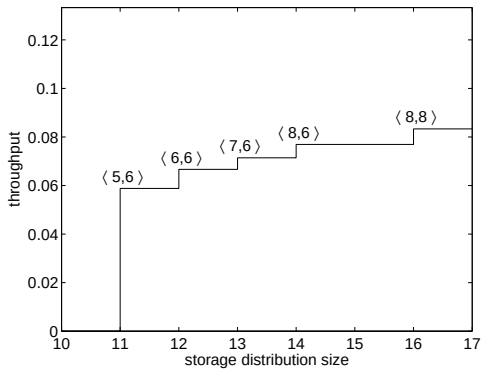


Fig. 9. Pareto space for CSDFG shown in Fig. 3.

Definition 18: (CAUSAL DEPENDENCY IN DEADLOCK) *In a deadlocked state, an actor a causally depends on an actor b via a channel α from b to a if and only if firing of a is prohibited by a lack of tokens on channel α.*

Based on this definition, we can also define a causal dependency graph for the deadlock case, which via Definitions 15 and 16 defines the storage dependencies for the deadlock case.

Definition 19: (CAUSAL DEPENDENCY GRAPH IN DEADLOCK) *Given a CSDFG (A_δ, C_δ) incorporating a storage distribution δ, with throughput zero (i.e., self-timed execution (eventually) deadlocks). The causal dependency graph (D, E) contains a node a for every actor $a \in A_\delta$. The set of dependency edges E contains an edge from a to b if and only if a causally depends on b in the deadlock state.*

In the remainder, 'dependency graph' refers to either the abstract causal dependency graph in case of CSDFGs with positive throughput and the causal dependency graph in deadlock otherwise.

IX. DESIGN-SPACE EXPLORATION

Sec. VII presents a technique to find the throughput for a given storage distribution and Sec. VIII provides the means to detect which channels are potentially limiting the graph's throughput. Using these techniques, it is possible to find the trade-offs between the distribution size and the throughput, i.e., the *Pareto space*. Fig. 9 shows this Pareto space for our example CSDFG. It shows that storage distribution $\langle 5, 6\rangle$ is the smallest distribution with a throughput larger than zero. The throughput of the graph can never go above $1/12$, as actor a always has, according to its entry in the repetition vector, to fire six times which requires 12 time steps. With a distribution size of 16 tokens, the maximal throughput can be achieved.

The following definition defines the minimal storage distributions that we are interested in.

Definition 20: (MINIMAL STORAGE DISTRIBUTION) *A storage distribution δ with throughput Th is minimal if and only if there is no storage distribution δ' with throughput Th' such that $|\delta'| \leq |\delta|$ and $Th' > Th$, or $|\delta'| < |\delta|$ and $Th' \geq Th$.*

Distributions $\langle 0, 0\rangle$, $\langle 5, 6\rangle$, $\langle 6, 6\rangle$, $\langle 7, 6\rangle$, $\langle 8, 6\rangle$ and $\langle 8, 8\rangle$ in the CSDFG of Fig. 3 are minimal, as shown in Fig. 9 (except for $\langle 0, 0\rangle$), but distribution $\langle 8, 7\rangle$ is not, because it yields the same throughput as configuration $\langle 8, 6\rangle$. Fig. 2 shows another example of this Pareto space, for the SDFG shown in Fig. 1. An interesting observation is that distributions $\langle 6, 2\rangle$ and $\langle 5, 3\rangle$ are both minimal, while having the same size. This example shows that in general multiple minimal storage distributions may exist with the same distribution size. Also note that the example Pareto spaces illustrate that the throughput is monotonically non-decreasing with the distribution size. Increasing a channel capacity can only lead to the earlier production of data tokens, and hence it can never lead to a decrease in throughput.

Algorithm 1 Find all minimal storage distributions

Input: A CSDFG G with maximal throughput Th_{max}
Result: A set P of pairs (storage distribution, throughput), containing precisely all minimal storage distributions

```
1:  procedure FINDMINSTORAGEDIST(G, Th_max)
2:      Let U be a list of unexplored storage distributions,
            ordered by size
3:      U ← [⟨0, ..., 0⟩]
4:      P ← ∅
5:      while no (δ, Th) ∈ P with Th = Th_max or, when a
6:        (δ, Th_max) ∈ P, there is some δ' ∈ U with |δ'| = |δ| do
7:          δ ← removeFirst(U)
8:          Create CSDFG G_δ which models δ in G
9:          Compute throughput Th
                    and dependency graph Δ of G_δ
10:         P ← P ∪ {(δ, Th)}
11:         Let S be the set of storage dependencies in Δ
12:         for each channel α in S do
13:             δ_n ← δ
14:             δ_n(α) ← δ(α) + step(α)
15:             U ← U ∪ {δ_n}
16:     Remove non-minimal storage distributions from P
```

Algorithm 1 presents the exploration algorithm for finding all minimal storage distributions, given a connected CSDFG G and its maximal throughput Th_{max}. This maximal throughput can be computed as the minimum throughput of all strongly connected components of the CSDFG, computed using the technique explained in Sec. VII. The algorithm uses a set U which contains all storage distributions that it needs to explore, ordered by their distribution size. Initially, the set U contains only the storage distribution $\langle 0, \ldots, 0\rangle$. The algorithm explores all storage distributions in U with smallest size first. A smallest storage distribution δ is removed from it and the algorithm computes the dependency graph Δ and throughput Th for this distribution. The distribution-throughput pairs are kept in a set P. The algorithm continues by constructing a new storage distribution δ_n for each channel α which has a storage dependency in Δ. In δ_n, the storage space of α is increased by the step size of the channel as explained in Sec. V. All other channels have the same storage space in δ_n as in δ. δ_n is then added to the set U. The main loop of the algorithm terminates when some storage distribution realizing the maximal throughput has been found and no other storage distributions of equal size remain to be explored. Finally, all non-minimal storage distributions are removed from P. Observe that this can be done via a single traversal through all the explored storage distributions stored in P by keeping this set sorted according to throughput. It is shown below that the algorithm returns precisely all minimal storage distributions.

To illustrate the workings of Algorithm 1, we discuss one iteration of the loop (line 5-15) while computing the throughput/storage trade-off space of the CSDFG of Fig. 3. Consider the

situation in which the algorithm is executing and it has reached line 5 while U contains the storage distribution $\langle 8, 6\rangle$ and P contains the storage distribution/throughput pairs that have already been explored (e.g., $(\langle 0, 0\rangle, 0)$, $(\langle 5, 6\rangle, 0.059)$, $(\langle 6, 6\rangle, 0.067)$ and $(\langle 7, 6\rangle, 0.071)$). The maximal throughput Th_{max} of the graph is equal to 0.083. The condition at lines 5-6 is true, since P contains no storage distribution/throughput pair that reaches Th_{max}. At line 7, the storage distribution $\langle 8, 6\rangle$ is removed from U. This storage distribution is modeled in the CSDFG of Fig. 3. The resulting CSDFG G_δ is shown in Fig. 5. Next, G_δ is executed to find its throughput and dependency graph. This dependency graph is the abstract dependency graph of Fig. 8. On line 10, the storage distribution/throughput pair $(\langle 8, 6\rangle, 0.77)$ is added to P. From the dependency graph, it follows that both the channels α and β have a storage dependency. Since the step size of α is one and the step size of β is two, lines 12-15 add the new storage distributions $\langle 9, 6\rangle$ and $\langle 8, 8\rangle$ to U. The algorithm then continues with the next iteration of the loop. In this iteration, it will explore the newly added storage distribution $\langle 9, 6\rangle$.

A few lemmas are needed to prove that the result of Algorithm 1 returns precisely all minimal storage distributions, or in other words, all throughput-buffering Pareto points for a CSDFG.

Lemma 1: Given a storage distribution δ_i with throughput Th_i. For any storage distribution $\delta_j \preceq \delta_i$ with throughput $Th_j < Th_i$ and dependency graph Δ_j, there is a channel α with a storage dependency in Δ_j such that $\delta_i(\alpha) > \delta_j(\alpha)$.

Proof: If δ_j is such that the graph deadlocks, the result is straightforward to prove using the dependency graph for the deadlock case. We present the case that Th_j is positive in more detail. In a self-timed execution, each actor firing has a causal dependency with at least one earlier actor firing (unless it is one of the first firings consuming the initial tokens). This gives chains of causal dependencies between all actor firings that occur during the execution. These chains of causal dependencies start with the initial firings of the graph, and either end at some point, or they are of infinite length. In a non-deadlocking self-timed execution, there must be at least one such chain of infinite length. (Otherwise, the graph would have delayed some firing unnecessarily.) Such infinite chains determine the throughput.

There is a finite number of states in the periodic phase and states in this phase are revisited each period. Hence, also the same causal dependencies are encountered again and again. So, each infinite chain of causal dependencies implies a cycle in the causal dependency graph as defined in Definition 14 and according to Theorem 2 also in the abstract causal dependency graph.

To increase the throughput, each of the causal dependency cycles must be broken. Since δ_i has a higher throughput than δ_j, we know that every causal dependency cycle of δ_j includes a storage dependency in Δ_j (because otherwise Th_j would be maximal, contradicting $Th_j < Th_i$). Since decreasing the capacity of channels can never increase throughput, to achieve throughput $Th_i > Th_j$, at least one storage dependency of each causal dependency cycle in Δ_j has been resolved by increasing the capacity of the corresponding channel. Hence, there must be some channel α with a storage dependency in Δ_j such that $\delta_i(\alpha) > \delta_j(\alpha)$. ■

Lemma 2: Given a storage distribution δ_i with throughput Th_i and a storage distribution δ_j such that $\delta_j \preceq \delta_i$ with throughput $Th_j < Th_i$. Then, Algorithm 1, from distribution δ_j, explores a storage distribution δ_k for which $\delta_j \preceq \delta_k \preceq \delta_i$, $|\delta_j| < |\delta_k|$ and $Th_j \leq Th_k \leq Th_i$.

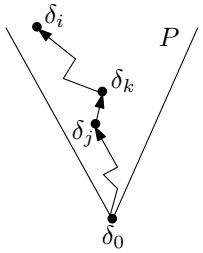


Fig. 10. Distributions reached by Algorithm 1.

Proof: Let S_j be the set of storage dependencies of δ_j. From Lemma 1, it follows that there exists a channel $\alpha \in S_j$ for which $\delta_j(\alpha) < \delta_i(\alpha)$. So, the capacity of α can be enlarged with at least one step before the storage space becomes equal to the storage space assigned to it in δ_i. Because $\alpha \in S_j$, Algorithm 1 does increase the storage space of α, which results in a new storage distribution δ_k. As (only) the storage space of α is increased, but not beyond its capacity in δ_i, it must hold that $\delta_j \preceq \delta_k \preceq \delta_i$ and $|\delta_j| < |\delta_k|$. From $\delta_j \preceq \delta_k \preceq \delta_i$, it also follows directly that $Th_j \leq Th_k \leq Th_i$. ■

Theorem 3: (CORRECTNESS OF ALGORITHM 1) *The set of all storage distributions contained in P which is constructed using Algorithm 1 contains precisely all minimal storage distributions.*

The proof of Theorem 3 is illustrated by Fig. 10.

Proof: For throughput 0, $\delta_0 = \langle 0, \ldots, 0\rangle$ is the (only) minimal storage distribution and it is always returned by the algorithm. Therefore, let δ_i be some minimal storage distribution with positive throughput Th_i. We must show that $\delta_i \in P$, i.e., that the algorithm will explore distribution δ_i. Initially, Algorithm 1 starts from the storage distribution $\delta_0 = \langle 0, \ldots, 0\rangle$ which satisfies the conditions of Lemma 2. Repeatedly using Lemma 2, we can show that the algorithm explores a series δ_k of storage distributions with $\delta_0 \preceq \delta_1 \preceq \delta_2 \preceq \ldots \preceq \delta_m \preceq \delta_i$ and for each k, $|\delta_k| < |\delta_{k+1}|$. From $\delta_k \preceq \delta_i$, it follows that $|\delta_k| \leq |\delta_i|$ and hence, after a finite number of m steps it must be that Lemma 2 no longer applies. Thus, it follows that $Th_m = Th_i$ and, because δ_i is minimal, that $|\delta_m| = |\delta_i|$. For a distribution $\delta_m \preceq \delta_i$ such that $|\delta_m| = |\delta_i|$, it must be that $\delta_m = \delta_i$, which shows that δ_i is explored by the algorithm. When the condition of the while loop no longer holds, no more minimal storage distributions can be reached from distributions in U.

So far, we have shown that set P contains all minimal storage distributions. However, it may also contain non-minimal distributions. The last line of the algorithm removes non-minimal storage distributions, completing the proof. ■

From the literature on dataflow graphs, lower bounds on the storage space required for each channel to avoid deadlock (i.e., throughput equal to zero) are known [11], [16]. These bounds can be used to speed up the initial phase of Algorithm 1. Distribution $\langle 0, \ldots, 0\rangle$ with all zero entries is by definition the only minimal storage distribution realizing zero throughput. Thus, to find all Pareto points with non-zero throughput, it is sufficient to start from the mentioned lower bounds.

An important and relevant question that remains is whether Algorithm 1 terminates. We show that if at least one actor has a bounded throughput, then Algorithm 1 ends. If all actors can increase their firing rate indefinitely, then there are infinitely many minimal storage distributions, and the algorithm cannot terminate.

Theorem 4: (TERMINATION) *For any connected CSDFG G that contains an actor with bounded throughput, Algorithm 1 terminates.*

Proof: Given a connected CSDFG G, Th_{max} is equal to the minimum of the throughputs of all strongly connected components in G. As there is at least one actor with bounded throughput, the throughput of the strongly connected component containing that actor must be bounded, and hence $Th_{max} < \infty$. The throughput Th_{max} is achievable within finite memory. This implies that there exists some storage distribution δ_{max} of finite size N that achieves throughput Th_{max}.

Algorithm 1 explores the storage distributions with increasing size. There is only a finite number of storage distributions of any size $n \leq N$. This implies that only a finite number of different storage distributions exist that have a size at most $|\delta_{max}|$. Within a finite number of steps, all distributions with size up to N are explored and a storage distribution with throughput Th_{max} is found, causing the algorithm to terminate. ■

The algorithm presented in this section is a modified version of the algorithm presented in [9]. These algorithms differ in the order in which storage distributions are explored and the termination condition (line 5) that is used. Algorithm 1 explores the storage distributions with increasing size. It ends immediately after it explored the storage distribution size of the minimal storage distributions that realize the maximal throughput. The algorithm in [9] searches the design space using a depth-first search algorithm and it ends when the list U of unexplored storage distributions is empty. The algorithm from [9] explores in this way all storage distributions that are explored by Algorithm 1. However, it may also explore storage distributions that are larger than the size of the minimal storage distributions that realize maximal throughput. In other words, the algorithm from [9] may explore more (but never less) storage distributions than Algorithm 1. Hence, the algorithm presented in this paper may terminate earlier than the algorithm from [9].

As mentioned in Sec. II, the problem of finding all minimal storage distributions of a CSDFG is NP-hard. Algorithm 1 has in fact an exponential worst-case complexity. This is due to the underlying throughput analysis technique that is used. This technique has an exponential worst-case complexity. Furthermore, the algorithm itself explores a set of storage distributions that can potentially also be exponentially large.

X. EXPERIMENTAL RESULTS

We performed a number of experiments on real DSP and multimedia application models to evaluate how our approach performs. The set of SDFG application models contains a modem [3], a satellite receiver [6], a sample-rate converter [3], an MP3-decoder [9] and an H.263 decoder [9]. We also included the often used bipartite SDFG from [3] in our SDFG benchmark. The set of applications modeled with a CSDFG contains an H.263 encoder [24], [25], a channel equalizer [26] and an MP3 playback application [7]. The CSDFG model formalized in Sec. III assumes that all port rate and execution time sequences are of equal length. To satisfy this requirement, the sequences given in the original CSDFGs must be concatenated. All sequences in these graphs have either a length 1 or a length N that is fixed for a given application ($N = 99$ in the H.263 encoder, $N = 8$ in the channel equalizer and $N = 39$ in the MP3 playback application). The execution times for the actors in all graphs were, when available, taken from the references. In other cases, they were obtained by analyzing the application source code with the worst-case execution time analysis technique described in [27]. For each of the graphs, the complete design space was explored. This resulted in a Pareto space showing the trade-offs between the throughput and distribution size for each graph.

The results of the experiments on the SDFGs are shown in Tab. I. The results for the CSDFGs are summarized in Tab. II. Both tables show the number of actors in each graph and the number of channels for which the buffers are being sized, the minimal distribution size for the smallest positive throughput, the maximum throughput that can be achieved and the distribution size needed to realize this throughput. They also show the number of Pareto points and the number of minimal storage distributions that were found during the design-space exploration. The results show that each Pareto point contains a single storage distribution.

For the SDFGs, an estimate on the number of storage distributions in the design space can be made. It is possible to compute an upper bound on the storage space required for each channel to achieve maximal throughput with finite channel capacities [2]. This upper bound and the lower bound, mentioned in Sec. IX and taken from [16], can be used to compute the number of different storage distributions in the design space (see row '#Distr. in space' of Tab. I). The next row shows the number of storage distributions explored by the algorithm. The results show that the algorithm explores only very few distributions from the space. For most SDFGs, it only explores the minimal storage distributions (excluding the trivial minimal storage distribution containing all zero entries, as explained in Sec. IX). This shows that the algorithm successfully prunes the design space. No good technique is known to accurately upper-bound the storage space requirements for channels in a CSDFG. Therefore, no good estimate can be made of the number of storage distributions in the design space of the CSDFGs. However, the results show that the number of storage distributions that is explored by the algorithm is also in this case limited.

The algorithm computes the throughput for each storage distribution it tries. This is done via a self-timed execution of the graph. The row 'Max. #states visited' shows the maximal number of different states that is visited during a throughput computation. Only a selected number of states must be stored (see Sec. VII) to compute the throughput of the graph. The maximal number of states that is stored is shown in the row 'Max. #states stored'.

All SDFGs, except the H.263 decoder, show a run-time in the order of milliseconds to explore the complete design space. The run-time for the H.263 decoder is large due to the large number of Pareto points contained in the space. The results on the CSDFGs show that the complete Pareto space can be computed within seconds for all applications. The MP3 playback CSDFG has the longest run-time due to the many Pareto points contained in its throughput/storage trade-off space.

It is interesting to consider the MP3 playback model of [7] in a bit more detail. In [7], a heuristic is presented that computes

TABLE I
EXPERIMENTAL RESULTS ON SDFGS.

	Bipartite	Sample rate	Modem	Satellite	MP3	H.263 decoder
#actors / #sized channels	$4/4$	$6/5$	$16/19$	$22/26$	$13/12$	$4/3$
Min. pos. throughput (s^{-1})	$4{\cdot}10^4$	$15{\cdot}10^4$	$3{\cdot}10^4$	$18{\cdot}10^4$	$7{\cdot}10^3$	50
Distr. size	28	32	38	1542	12	4753
Max. throughput (s^{-1})	$6{\cdot}10^4$	$17{\cdot}10^4$	$6{\cdot}10^4$	$23{\cdot}10^4$	$8{\cdot}10^3$	100
Distr. size	35	34	40	1544	16	8006
#Pareto points	9	4	4	3	4	3255
#Min. distr.	9	4	4	3	4	3255
#Distr. in space	$1{\cdot}10^8$	$9{\cdot}10^{12}$	$1{\cdot}10^{10}$	$2{\cdot}10^{65}$	4096	$3{\cdot}10^{10}$
#Distr. checked	51	3	4	4	7	$292{\cdot}10^3$
Max. #states visited	652	$6{\cdot}10^6$	134	10377	33579	$8{\cdot}10^6$
Max. #states stored	20	5328	2	241	212	1124
Exec. time	$1ms$	$1ms$	$2ms$	$7ms$	$2ms$	$53min$

a storage space distribution under a throughput constraint. The objective is to minimize the size of the storage distribution. The reported results state that the heuristic can compute within the order of 10^{-2}s a storage distribution which is 5% larger than the smallest storage distribution allowing maximal throughput. Using our algorithm, the optimal solution can be found in 26s (see Tab. II). Depending on the context in which buffer sizing is applied, this run-time may or may not be acceptable. In general, however, the exponential worst-case complexity of our technique could potentially lead to prohibitively large run-times. In those cases, our technique can be combined with any heuristic for buffer sizing. For example, the heuristic from [7] can be used to compute a storage distribution close to the optimum. A fraction of the storage space computed for each channel by the heuristic can then be used as a starting distribution in (line 3 of) our algorithm. In this way, our algorithm explores the trade-off space just below the storage distribution computed by the heuristic for a smaller distribution that still satisfies the throughput constraint. To test this approach, we ran our algorithm on the MP3 playback CSDFG with the initial storage distribution equal to 90% of the storage distribution requirements computed by the heuristic from [7] for this CSDFG. Our algorithm was able to find the optimal storage distribution within 1s. This illustrates how our algorithm can be combined with a heuristic, which will in general improve the results of the heuristic with little effort.

TABLE II
EXPERIMENTAL RESULTS ON CSDFGS.

	H.263 encoder[1]	Channel equalizer	MP3 playback
#actors / #sized channels	$6/6$	$12/20$	$4/2$
Min. pos. throughput (s^{-1})	$8{\cdot}10^{-2}$	3	4.0
Distr. size	104	19	921
Max. throughput (s^{-1})	0.3	3	8.3
Distr. size	105	19	1842
#Pareto points	3	2	829
#Min. distr.	3	2	829
#Distr. checked	2	1	2296
Max. #states visited	$12{\cdot}10^6$	2298	14231
Max. #states stored	2	2	4
Exec. time	$10s$	$4ms$	$26s$

[1]The (unexpectedly low) throughput values for this model are obtained when assuming a 500 Mhz processor and using the cycle counts reported in [24] (which does not specify the used platform).

XI. APPROXIMATION OF BUFFER SIZES

A. A Generic Approximation Technique

The experimental results of the previous section show that the search space of distributions is pruned efficiently by looking for storage dependencies. Nevertheless, the number of distributions that need to be explored may still be large, potentially leading to long run-times of the algorithm. An approximation of the exact result can be obtained by reducing the number of distributions that need to be explored, for instance by changing the step size for increasing the channel capacities. We have shown that for a channel α considering as sizes all multiples of $step(\alpha)$ guarantees that all minimal distributions are found. In this section, we consider exploring only a set K_α of capacities for channel α and for any capacity k, we use $\lceil k \rceil^{K_\alpha}$ to denote the smallest capacity in K_α which is at least k. We require that sets K_α be such that such capacity always exists (i.e., that channel capacities can always be increased). Concrete examples of such sets, that we also use for our experimental evaluation later in this section, are sets K_α^n, for any number $n \in \mathbb{N}$, defined as $\{k \in \mathbb{N} \mid k = n \cdot step(\alpha)\}$, i.e., only multiples of the $step(\alpha)$ are considered for the given multiplication factor n.

Algorithm 1 is adapted as follows. Line 13 becomes:

$$\delta_n(\alpha) \leftarrow \lceil \delta(\alpha) + 1 \rceil^{K_\alpha}.$$

That is, the next smallest capacity in the given set K_α is chosen. We can prove the following property of the adapted algorithm. The adapted algorithm finds all minimal storage distributions δ among all distributions with channel capacities $\delta(\alpha) \in K_\alpha$ for all $\alpha \in C$. The property is proved using Lemmas 1 and 2 and Theorem 3, while restricting attention to distributions within the limited set.

From the fact that the adapted algorithm finds all minimal storage distributions among the reduced set, we can derive the following bound on the discrepancy of the result from the optimal result.

Theorem 5: (OVERESTIMATION BOUND) *For every minimal storage distribution δ with throughput Th, found in the full search, there is a storage distribution δ' with throughput Th' that is minimal in the reduced search space such that $Th' \geq Th$ and $|\delta'| \leq \sum_{\alpha \in C} \lceil \delta(\alpha) \rceil^{K_\alpha}$.*

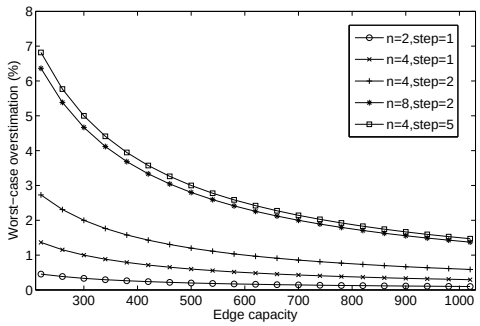


Fig. 11. Worst-case overestimation for various channel capacities, step sizes, and multiplication factors.

Proof: An increase in channel capacity cannot decrease the throughput. If we round all channel capacities in δ up to $\lceil\delta(\alpha)\rceil^{K_\alpha}$, we obtain a distribution with throughput Th', at least throughput Th. If this distribution is minimal in the reduced search space, the theorem is proved. If it is not minimal, there exists a minimal one with the same throughput and smaller distribution size, or with a higher throughput and the same size. In both cases, this distribution satisfies the theorem. ■

Note that the adapted algorithm still has an exponential worst-case complexity. However, it allows a trade-off between run-time and quality of the end result, by appropriately choosing the K_α. One can choose, for example, the sets K_α^n already mentioned above, possibly with different multiplication factors per channel. Theorem 5 then gives a bound on the worst-case loss in quality (buffer-size overestimation). Consider a single channel α. Theorem 5 and the definition of K_α^n imply that

$$\delta'(\alpha) \leq \lceil\delta(\alpha)\rceil^{K_\alpha^n} \leq \delta(\alpha) + (n-1)step(\alpha).$$

It follows that the relative overestimation for α is bounded as follows:

$$\frac{\delta'(\alpha) - \delta(\alpha)}{\delta(\alpha)} \leq \frac{(n-1)step(\alpha)}{\delta(\alpha)}, \quad (1)$$

which is the expected result that the overestimation per channel can be at most $n-1$ times the step size.

Figure 11 shows the worst-case overestimation for a single channel, for various channel capacities, step sizes, and multiplication factors. The channel capacities and step sizes are in line with those observed in the models of our benchmark. The relative worst-case overestimation is small for increasing channel size, which is when the approximation algorithm is most useful, because only for large distribution spaces the run-time of our exact technique may become problematic. Note that the relative worst-case overestimation does not change when considering multiple channels (assuming the same capacities, step sizes, and multiplication factors). The likelihood that the worst case will occur in fact decreases with increasing numbers of channels in the CSDFG. Another way to limit the over-estimation is to search the trade-off space with several multiplication factors per channel, as illustrated below.

It is also possible to try to bound the relative overestimation via an appropriate choice of K_α, by choosing for K_α the set $\{\lceil(1+q)^n\rceil step(\alpha) \mid n \in \mathbb{N}\}$, for some appropriately small q. Except for rounding effects, this choice for the K_α limits the overestimation to $q \cdot 100\%$.

B. Experimental Results

The experimental results presented in Sec. X show that design-space exploration of the H.263 decoder and the MP3 playback application take the most time from all the tested models. For both applications, this is due to the large number of Pareto points in the trade-off space. However, the throughput of most of the Pareto points is close to each other. In practice, it is not interesting to find all these points. The approximation technique presented in the previous section can be used to reduce the number of different storage distributions that is explored.

Experiments have been performed with this approximation algorithm on the H.263 decoder and MP3 playback application. For both applications, a uniform multiplication factor has been used for the different channels. The step size in the H.263 decoder was multiplied with a factor of 3, 9, and 27. For the MP3 playback application, a multiplication factor of 3, 5, and 15 has been used. The results for these experiments are shown in Tab. III. (The last column in the table is explained below.) The results show that the approximation technique drastically improves the run-time of the exploration at the cost of a reduced number of Pareto points found (but as already said, it is hard to imagine that hundreds or thousands of Pareto points are practically meaningful).

The approximation algorithm may lead to an overestimation of the required storage space for a given throughput. The table shows for each experiment the maximum overestimation observed for an arbitrary Pareto point in the complete trade-off space, the average overestimation over the entire space, and the overestimation of the minimal buffer requirements allowing maximal throughput. The results show that the overestimation is very small in general, which shows that dropping Pareto points for these models does not have much impact in terms of storage requirements computed for a given throughput constraint. The peak in maximum overestimation for the largest multiplication factor in the MP3 experiment is to be expected, because large overestimation may occur for large multiplication factors in combination with small buffer sizes. However, small buffers are the part of the trade-off space that our algorithm explores first. In those cases, the exact technique, an approximation with a smaller multiplication factor, or an approximation aiming to bound the relative overestimation (as explained at the end of the previous subsection) can be used to determine the appropriate buffer sizes satisfying the throughput constraint.

The experiments reported in Tab. III use only one multiplication factor, which is the same for all the sized channels. There are several ways to improve the obtained results. One way is to carefully select different multiplication factors for the different channels. Another way is to apply our exact technique (or another approximation) on a designated part of the approximated space, in the same way as the combination of our technique with heuristics that we explained earlier. Given a throughput constraint, one can first make a coarse approximation of the trade-off space with a large multiplication factor. Then, one can choose a distribution that comes close to satisfying the constraint as the starting point for a finer grain approximation or an exact exploration of the trade-off space up to the point that the throughput constraint is satisfied.

A third way to improve approximation results is to simply combine the results of two or more approximations of the trade-off space. This may lead to a reduced overestimation, as illustrated by the last column of Tab. III. Combining the approximations of

TABLE III
RESULTS ON APPROXIMATION ALGORITHM.

	H.263 decoder				MP3 playback				
	exact	$n=3$	$n=9$	$n=27$	exact	$n=3$	$n=5$	$n=15$	$n=3,5$
Min. pos. throughput (s^{-1})	50	50	50	50	4.0	4.0	4.0	4.0	4.0
Distr. size	4753	4753	4753	4753	921	921	921	921	921
Max. throughput (s^{-1})	100	100	100	100	8.3	8.3	8.3	8.3	8.3
Distr. size	8006	8006	8012	8021	1842	1842	1846	1851	1842
#Pareto points	3255	1087	365	124	829	278	186	43	336
#Min. distr.	3255	1087	365	124	829	278	186	43	336
#Distr. checked	$292{\cdot}10^3$	28720	3613	558	2296	357	185	42	542
Max. overest.	-	0.07%	0.24%	0.69%	-	3.72%	7.33%	26.91%	3.72%
Avg. overest.	-	0.03%	0.10%	0.33%	-	0.17%	1.51%	5.28%	0.14%
Min.buf. / max.thr. overest.	-	0%	0.07%	0.19%	-	0%	0.22%	0.49%	0%
Exec. time	$53min$	$5min$	$36s$	$7ms$	$26s$	$4s$	$2s$	$0.49s$	$6s$

the trade-off space obtained via multiplication factors 3 and 5 leads to an increased number of Pareto points found, resulting in a reduced average overestimation when compared to the two approximations in isolation. We obtained the results reported in Tab. III by explicitly computing the two approximations, and then combining the results. A more efficient implementation would first compute the approximation with the largest multiplication factor, and then use the information about distributions already explored while computing the approximation with the smaller multiplication factor. This would lead to an amount of checked distributions and a run-time which are less than the sums of those values for the individual approximations. The experiment shows the versatility of the approximation technique. Note that it does not make sense to combine approximations when one multiplication factor is a divisor of (one of) the other multiplication factor(s) (which is why Tab. III does not report any other combinations).

C. Scalability

The experimental results presented so far show that it is possible to explore the trade-off space for all the application models in our benchmark within seconds, either via the exact technique or by approximation (with only very little overestimation). However, both the exact algorithm and the approximation technique have an exponential worst-case complexity. The observed run-times for the exact exploration indicate that the run-times may become problematic when models and/or Pareto spaces grow in size, while the approximation technique is fast in the two tested cases. To investigate scalability, we adapted the SDF^3 toolkit [28] to generate three synthetic CSDFGs, for which the exact algorithm does not terminate in several hours, confirming that the run-times of the exact algorithm may become very large. However, it is important to note that these graphs differ in a number of aspects from the realistic application models in our benchmark. Realistic applications are often a relatively straightforward pipeline of actors with only one or a few cycles. Furthermore, the number of branches in the graphs is limited. In the generated graphs, there are many cycles and branches with complex interactions. As a result, the number of Pareto points in these generated graphs is very large. Consequently, many different storage distributions have to be explored, which causes the run-time of the exact algorithm to become very large.

Since the approximation technique yields a reduction of the explored space that is exponential in terms of the number of channels being sized, it is interesting to test the scalability of the approximation technique on these artificial models. We tested the approximation with multiplication factors 3, 4 and 5 for all channels in the graphs. These channels all have a step size of one. Tab. IV reports the results, showing run-times and, using Eqn. 1, the calculated upper bound on the overestimation for the minimal buffer sizes needed for maximal throughput. The results show that the approximation algorithm can successfully prune the design space, even for extremely large trade-off spaces, guaranteeing that the overestimation stays within very tight bounds.

TABLE IV
EXPERIMENTAL RESULTS ON SYNTHETIC CSDFGS.

	$n=3$	$n=4$	$n=5$
graph 1	50s / 0.38%	16s / 0.38%	7s / 0.37%
graph 2	161s / 0.91%	70s / 0.91%	29s / 0.91%
graph 3	271s / 0.71%	45s / 0.71%	28s / 0.71%

XII. CONCLUSION

We have presented a method to explore the trade-offs between the throughput and buffering requirements for SDFGs and CSDFGs. It generalizes the techniques from [9] to CSDFGs. It also improves the exploration algorithm from [9] allowing better pruning of the trade-off space. The experiments show that, despite the complexity of the problem, it is possible to perform an exact design-space exploration for real application kernels. We also show that our technique can be combined with existing heuristics for buffer sizing. This makes it possible to compute sharper bounds on the buffer requirements than those found with the heuristic alone with limited run-time overhead. It may in fact often lead to optimal results if the applied heuristic yields a sufficiently accurate estimate as the starting point for our algorithm.

In addition to the exact exploration algorithm, we presented a generic and very versatile approximation technique based on the exact algorithm. The approximation provides throughput guarantees, and it has a proven analytical upper bound on the overestimation in buffer sizes. Approximation of the trade-off space can be used when the run-times of the exact technique would become problematic. The results for the approximation technique show that it can drastically improve the run-time needed for the exploration of the trade-off space with only very limited overestimation of the storage space.

The techniques presented in this paper are implemented in the freely available SDF^3 toolkit [28]. We use the techniques in a predictable multiprocessor design flow [29] based on the dataflow

model of computation, integrating the results with the processing and communication resource allocation techniques of [30].

REFERENCES

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

[2] E. Lee and D. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers*, vol. 36, no. 1, pp. 24–35, January 1987.

[3] S. Bhattacharyya, P. Murthy, and E. Lee, "Synthesis of embedded software from synchronous dataflow specifications," *Journal on VLSI Signal Processing Systems*, vol. 21, no. 2, pp. 151–166, June 1999.

[4] K. Denolf, A. Chirila-Rus, P. Schumacher, R. Turney, K. Vissers, D. Verkest, and H. Corporaal, "A systematic approach to design low-power video codec cores," *EURASIP Journal on Embedded Systems*, vol. 2007, no. 3, pp. 1–14, March 2007.

[5] P. Poplavko, T. Basten, M. Bekooij, J. van Meerbergen, and B. Mesman, "Task-level timing models for guaranteed performance in multiprocessor networks-on-chip," in *Compilers, Architecture, and Synthesis for Embedded Systems, CASES 03, Proceedings*. ACM, 2003, pp. 63–72.

[6] S. Ritz, M. Willems, and H. Meyr, "Scheduling for optimum data memory compaction in block diagramoriented software synthesis," in *Acoustics, Speech, and Signal Processing, ICASSP 95, Proceedings*. IEEE, 1995, pp. 2651–2654.

[7] M. Wiggers, M. Bekooij, and G. Smit, "Efficient computation of buffer capacities for cyclo-static dataflow graphs," in *Design Automation Conf., DAC 07, Proceedings*. ACM, 2007, pp. 658–663.

[8] 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 *Application of Concurrency to System Design, ACSD 06, Proceedings*. IEEE, 2006, pp. 25–36.

[9] S. Stuijk, M. Geilen, and T. Basten, "Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs," in *Design Automation Conf., DAC 06, Proceedings*. ACM, 2006, pp. 899–904.

[10] P. Murthy, "Scheduling techniques for synchronous multidimensional synchronous dataflow," Ph.D. dissertation, UC Berkeley, 1996.

[11] M. Adé, R. Lauwereins, and J. Peperstraete, "Data minimisation for synchronous data flow graphs emulated on DSP-FPGA targets," in *Design Automation Conf., DAC 97, Proceedings*. ACM, 1997, pp. 64–69.

[12] S. Bhattacharyya, P. Murthy, and E. Lee, *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.

[13] M. Geilen, T. Basten, and S. Stuijk, "Minimising buffer requirements of synchronous dataflow graphs with model-checking," in *Design Automation Conf., DAC 05, Proceedings*. ACM, 2005, pp. 819–824.

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

[15] C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu, "A formal approach to the scheduling problem in high-level synthesis," *IEEE Transactions on Computer-Aided Design*, vol. 10, no. 4, pp. 464–475, April 1991.

[16] P. Murthy and S. Bhattacharyy, "Shared memory implementations of synchronous dataflow specifications," in *Design Automation and Test in Europe, DATE 00, Proceedings*. IEEE, 2000, pp. 404–410.

[17] Q. Ning and G. Gao, "A novel framework of register allocation for software pipelining," in *Symposium on Principles of Programming Languages, Proceedings*. ACM, 1993, pp. 29–42.

[18] H. Oh and S. Ha, "Efficient code synthesis from extended dataflow graphs," in *Design Automation Conf., DAC 02, Proceedings*. ACM, 2002, pp. 275–280.

[19] K. Altisen, G. Gößler, and J. Sifakis, "A methodology for the construction of scheduled systems," in *Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 00, Proceedings, volume 1926 in LNCS*. Springer-Verlag, 2000, pp. 106–120.

[20] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi, "Times: a tool for schedulability analysis and code generation of real-time systems," in *Formal Modeling and Analysis of Timed Systems, FORMATS 03, Proceedings, volume 2791 in LNCS*. Springer-Verlag, 2003, pp. 60–72.

[21] S. Shukla and R. Gupta, "A model checking approach to evaluating system level dynamic power management policies for embedded systems," in *High-Level Design Validation and Test Workshop, HLDVT 01, Proceedings*. IEEE, 2001, pp. 53–57.

[22] A. Ghamarian, M. Geilen, T. Basten, B. Theelen, M. Mousavi, and S. Stuijk, "Liveness and boundedness of synchronous data flow graphs," in *Formal Methods in Computer Aided Design, FMCAD 06, Proceedings*. IEEE, 2006, pp. 68–75.

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

[24] D. Kim, "System-level specification and cosimulation for multimedia embedded systems," Ph.D. dissertation, Seoul National University, 2003.

[25] H. Oh and S. Ha, "Fractional rate dataflow model for efficient code synthesis," *Journal of VLSI Signal Processing*, vol. 37, no. 1, pp. 41–51, May 2004.

[26] A. Moonen, M. Bekooij, R. van den Berg, and J. van Meerbergen, "Evaluation of the throughput computed with a dataflow model - a case study," TU Eindhoven, Tech. Rep., March 2007.

[27] S. Gheorghita, S. Stuijk, T. Basten, and H. Corporaal, "Automatic scenario detection for improved WCET estimation," in *Design Automation Conf., DAC 05, Proceedings*. ACM, 2005, pp. 101–104.

[28] S. Stuijk, M. Geilen, and T. Basten, "SDF^3: SDF For Free," in *Application of Concurrency to System Design, ACSD 06, Proceedings*. IEEE, 2006, pp. 276–278, SDF^3 is available via www.es.ele.tue.nl/sdf3.

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

[30] S. Stuijk, T. Basten, M. Geilen, and H. Corporaal, "Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs," in *DAC 07, Proceedings*. ACM, 2007, pp. 777–782.

Sander Stuijk received his Master's degree (with honors) in Electrical Engineering in 2002 and his Ph.D. degree in 2007 from the Eindhoven University of Technology. He is currently a postdoc in the Department of Electrical Engineering at the Eindhoven University of Technology. His research focuses on the mapping of streaming multimedia applications on multiprocessor platforms.

Marc Geilen received his Master's degree (with honors) in Information Technology in 1996 and his Ph.D. in 2002, from the Eindhoven University of Technology. He is currently an assistant professor in the Department of Electrical Engineering and has been involved with different European and national research projects. His research interests include validation and (formal) verification, modeling, simulation and programming paradigms for streaming multimedia systems, multiprocessor platforms and wireless sensor networks, and multi-dimensional optimization and trade-off analysis. He (co-)authored publications on these topics and he is a member of technical program committees and a topic chair in the DATE 2008 program committee.

Twan Basten is an associate professor in the Department of Electrical Engineering at the Eindhoven University of Technology. He has a Master's degree (with honors) and a Ph.D. degree in Computing Science from the same university. Twan Basten worked as visiting researcher at the University of Waterloo, Canada, Philips Research Laboratories, Eindhoven and Carnegie Mellon University, Pittsburgh, PA. His research interest is the design of complex, resource-constrained embedded systems, based on a solid mathematical foundation, with a focus on multiprocessor systems and models of computation. Twan Basten was the Ambient Intelligence co-chair in the DATE 2003 PC, topic chair in the DATE 2004 and 2005 PCs, and the PC co-chair for ACSD 2007. He (co)authored over 80 scientific publications.