Exploring Trade-Offs in Buffer Requirements and Throughput Constraints for 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 Synchronous Dataflow Graphs. Communication between nodes in these graphs requires storage space which influences throughput. We present exact techniques to chart the Pareto space of throughput and storage trade-offs, which can be used to determine the minimal storage space needed to execute a graph under a given throughput constraint. The feasibility of the approach is demonstrated with a number of examples.

Categories and Subject Descriptors: C.3 [Special-purpose and Application-based Systems] Signal processing systems

General Terms: Algorithms, Experimentation, Theory.

Keywords: Synchronous Dataflow, buffering, throughput, optimization.

1. INTRODUCTION

Synchronous Dataflow Graphs (SDFGs, [11]) have traditionally been used to model and analyze sequential DSP applications. Recently, they are also used for designing and analyzing multimedia applications realized using multiprocessor systems-on-chip. The main focus is on predicting the timing behavior of these complex media systems.

The nodes in an SDFG communicate data with each other. Storage space, *buffers*, must be allocated for this data. At design time, the allocation (size) of this storage space must be determined. The available storage space in an embedded system is usually very limited. Therefore, the storage space allocated for the graph should be minimized. Minimizing storage has the additional advantage that it saves energy.

Minimization of buffer requirements in SDFGs has been studied before, see for example [1, 4, 5, 7, 9, 10, 12, 13, 14]. The proposed solutions target mainly single-processor systems. Modern media applications, however, often target multi-processor systems. 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 [1, 4, 5, 7, 12, 14] 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 [9], 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 [10]. 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. In [13], buffer minimization for maximal throughput of a subclass of SDFGs (homogeneous SDFGs) is achieved via an integer linear programming approach. In general, the minimal buffer sizes obtained with this approach cannot be translated to exact minimal buffer sizes for arbitrary SDFGs. We propose, in contrast to existing work, an exact technique to determine all trade-offs (Pareto points) between the throughput and buffer size for an SDFG.

The buffer minimization problem is known to be NP-complete [4]. Researchers have successfully applied explicit state-space exploration techniques to solve NP-complete (and even worse) scheduling problems [2, 3, 17]. In the context of buffer minimization, [7] proposed a state-space exploration technique to find minimal buffer requirements to execute an SDFG with a deadlock-free schedule. This has motivated us to apply a state-space exploration based technique to the problem of storage-throughput trade-off analysis. However, our technique is in general not exhaustive. It prunes the search space in an efficient way, as confirmed by our experiments, without loosing any Pareto points.

We first introduce the timed SDF model and formalize the storage requirements and throughput of an SDFG. Subsequently, we explain our technique to perform a design-space exploration to find the trade-offs between throughput and memory consumption of an SDFG. The feasibility of the approach is demonstrated with a number of case studies. For space reasons, proofs are published separately in [19].

2. SYNCHRONOUS DATAFLOW GRAPHS

An example of a Synchronous Dataflow graph (SDFG) is depicted in Fig. 1. The nodes of an SDFG are called *actors*; they represent functions that are computed by reading *tokens* (data items) from their input ports, and writing the results of the computation as tokens on the output ports. An essential property of SDFGs is that every time an actor *fires* (performs a computation) it consumes the same amount of tokens from its input ports and produces the same amount of tokens on its output ports. These amounts are called the port *rates* and are visualized as port annotations. Actor firings are atomic and require a fixed execution time, denoted with a number in the actors. The edges in the graph, called *channels*, represent data that is communicated from one actor to another. The channels may contain tokens, depicted with a black dot and an attached number defining the number of tokens present in the channel. The storage space

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

DAC 2006, July 24–28, 2006, San Francisco, California, USA.

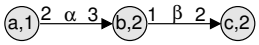

Figure 1: Example SDFG.

of a channel is in principle unbounded, i.e., it can contain arbitrarily many tokens. Formally an SDFG is defined as follows. We assume a set $Ports$ of ports, and with each port $p \in Ports$ we associate a finite rate $Rate(p) \in \mathbb{N}\backslash\{0\}$.

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

Definition 2. (SDFG) *An SDFG is a tuple (A, C) consisting of a finite set A of actors and a finite set $C \subseteq Ports^2$ 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, and all channels are connected to ports of some actor. For every actor $a = (I, O, \tau) \in A$, we denote the set of all channels that are connected to the ports in I (O) by $InC(a)$ ($OutC(a)$).*

As mentioned, actor execution is defined in terms of *firings*. When an actor a starts its firing, it removes $Rate(q)$ tokens from all $(p, q) \in InC(a)$. The execution continues for $\tau(a)$ time units and when it ends, it produces $Rate(p)$ tokens on every $(p, q) \in OutC(a)$. It is possible to choose rates in an SDFGs such that the SDFG deadlocks or such that tokens accumulate on the channels. In the latter case an SDFG can only execute in unbounded memory. Consistency [11] is known to be a necessary condition to allow an execution within bounded memory. Since consistency is straightforward to check, we focus on consistent SDFGs. Furthermore, we assume connectedness. For unconnected graphs, analysis can be done per connected subgraph.

3. OPERATIONAL SEMANTICS OF SDF

SDFG 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. Channels have infinite storage space, which means that there is always sufficient space available for output. We abstract from the actual data that is being communicated or processed by actors and treat all data elements equally in the form of (normalized) 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 an SDFG, we need to keep track of the distribution of tokens over the channels, of the start and end of actor firings, and the progress of time.

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

Definition 3. (Channel quantity) *A channel quantity on the set C of channels is a mapping $\gamma : C \to \mathbb{N}$. 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 $c \in C_1$, $\gamma_1(c) \leq \gamma_2(c)$. 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 a firing of some actor a can be described by a channel quantity $Rd(a) = \{(p, Rate(p)) | p \in In(a)\}$ and the amount of tokens produced at the end of a firing by a channel quantity $Wr(a) = \{(p, Rate(p)) | p \in Out(a)\}$.

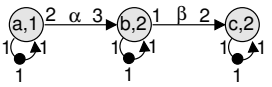

Figure 2: Limited auto-concurrency.

Definition 4. (State) *The state of an SDFG (A, C) is a pair (γ, υ). 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 $\upsilon : A \to \mathbb{N}^{\mathbb{N}}$ associates with each actor $a \in A$ a multiset of numbers representing the remaining times of different firings of a. We assume that the initial state of an SDFG is given by some initial token distribution γ, which means the initial state equals $(\gamma, \{(a, \{\}) \mid a \in A\})$ (with $\{\}$ denoting the empty multiset).*

The use of a multiset of numbers to keep track of actor progress instead of a single number allows multiple simultaneous firings of the same actor (auto-concurrency). This is in line with the standard SDF semantics. 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 with a single token to all actors as shown in Fig. 2. The dynamic behavior of the SDFG 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 5. (Transition) *A transition of SDFG (A, C) from state (γ_1, υ_1) to state (γ_2, υ_2) is denoted by $(\gamma_1, \upsilon_1) \xrightarrow{\beta} (\gamma_2, \upsilon_2)$ where label $\beta \in (A \times \{start, end\}) \cup \{clk\}$ denotes the type of transition.*

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

Definition 6. (Execution) *An execution of an SDFG is an infinite alternating sequence of states and transitions $s_0 \xrightarrow{\beta_0} s_1 \xrightarrow{\beta_1} \ldots$ from some designated initial state s_0.*

Note that even a deadlocked SDFG (no actor is firing or ready to fire) has an infinite execution as time always progresses.

4. STORAGE REQUIREMENTS

As mentioned in Sec. 2, channels have unbounded storage space. 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 an SDFG 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 [12]. 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 [1, 5]. Hybrid forms of both options can be used [7]. In this paper, we assume channels cannot share memory space. This gives a conservative bound on the required memory space when the SDFG is implemented using shared memory. In that case, the SDFG may require less memory, but it will never require more memory than determined by our method.

The maximum number of tokens which can be stored in a channel (*channel capacity*) is called a *storage distribution*.

Definition 7. (Storage distribution) *A storage distribution of an SDFG (A, C) is a channel quantity δ that associates with every $c \in C$, the capacity of the channel.*

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

Definition 8. (Distribution size) *The size of a storage distribution δ is given by: $|\delta| = \sum_{c \in C} \delta(c)$.*

A possible storage distribution for the SDFG shown in Fig. 1 would be $\delta(\alpha) = 4$ and $\delta(\beta) = 2$, denoted as $\langle \alpha, \beta \rangle \mapsto \langle 4, 2 \rangle$. It has a distribution size of 6 tokens.

In an SDFG state, a channel (p, q) from actor a to actor b does not contain an arbitrary number of tokens. Assume that the channel contains in the initial state of the execution d tokens. After n firings of a and m firings of b, the channel contains $n \cdot Rate(p) - m \cdot Rate(q) + d$ tokens. This is equal to $k \cdot \gcd(Rate(p), Rate(q)) + d \bmod \gcd(Rate(p), Rate(q))$ tokens with $k = (n \cdot Rate(p) - m \cdot Rate(q) + d) \div \gcd(Rate(p), Rate(q))$. The number of tokens in a channel, and hence the storage space which can be used usefully, depends via k on the gcd of the rate at which the actors a and b produce and consume tokens. This gcd is called the *step size* of the channel.

The bound on the storage space of each channel can be modeled in an SDFG (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 with $Rate(p) = Rate(p_\delta)$ and $Rate(q) = Rate(q_\delta)$. The number of initial tokens on the channel (q_δ, p_δ) models the storage space of the channel (p, q). Subscript 'δ' denotes elements used to model storage space. The SDFG which models the storage distribution δ in an SDFG (A, C) is denoted (A_δ, C_δ). Fig. 3 shows the SDFG which encodes the storage distribution $\langle 4, 2 \rangle$ 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.

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,

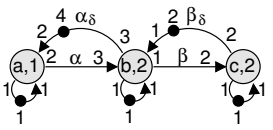

Figure 3: SDFG with storage distribution $\langle 4, 2 \rangle$.

the model assumes that space to produce output tokens is available when an actor starts firing and the 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 written earlier.

5. 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, which depends on the execution of the SDFG. There exists one type of execution, namely self-timed execution [18], which gives maximal throughput. 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.

Definition 9. (Throughput) *The throughput of an actor a for the self-timed execution σ of an SDFG is defined as the average number of firings of a per time unit in σ.*

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 connected SDFG incorporating a storage distribution (which is by definition finite) is always strongly connected. In that situation, the fixed rates of the actor ports ensure that the number of times actors fire with respect to each other (repetition vector [6]) is constant. In other words, the throughput of each pair of actors in a graph is related to each other via a constant. In the remainder, we assume that some arbitrary actor has been selected to compute throughput.

To compute the throughput of actor c of our example (see Fig. 3), we first look at the transition system of the self-timed execution as shown in Fig. 4. All clock transitions are shown explicitly. Between clock transitions, there can be multiple start and/or end transitions enabled simultaneously. These start and end transitions are independent of each other. Independent of the order in which they are applied, the final state before each clock transition, and the first state after each clock transition, are always the same. Therefore, all start and end transitions are shown as one annotated step. 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*. In our example, actor c ends its firing for the first time after 9 clock transitions in the gray state. At that moment, the actor is in the periodic phase of the schedule and fires each 7 time units. The periodic phase is repeated indefinitely. Hence, the average time between two firings over the whole schedule converges to the average time between two firings in the periodic phase. So, the throughput of c is $1/7$.

6. THROUGHPUT CALCULATION

Theorem 1. *The state-space for any SDFG (A, C) with storage distribution δ contains always exactly one cycle.*

The theorem states that the state-space of any SDFG with bounded storage space for all channels consists of a transient phase followed by a periodic phase. Def. 9 defines the

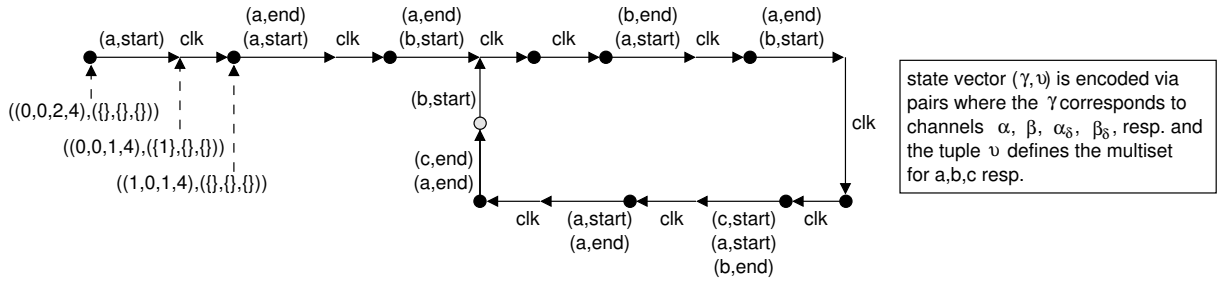

Figure 4: SDF state space of the example SDFG.

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. *The throughput of an actor a in an SDFG 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.*

The throughput of an actor in an SDFG can be computed by executing the SDFG 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 can be computed using Proposition 1. The number of states that must be remembered can be reduced. We can enforce deterministic execution by choosing a fixed order among simultaneously enabled transitions in the transition system without affecting the throughput. Thus to detect the cycle, only the states that represent the end of a firing of some arbitrary actor must be kept. 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 (see Fig. 4), only the gray state must be stored as this is the only state in which a firing of c ends. More details on throughput analysis for SDFGs can be found in [8].

7. STORAGE DEPENDENCIES

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

DEFINITION 10. (CAUSAL DEPENDENCY) *A firing of an actor a causally depends on the firing of an actor b via a channel c if the firing of a consumes a token from c produced by the firing of b on c 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 the 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.

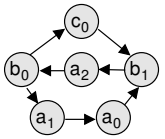

Figure 5: Dependency graph of the example SDFG.

DEFINITION 11. (DEPENDENCY GRAPH) *Given an SDFG (A_δ, C_δ) incorporating a storage distribution δ and a sequence of states and transitions p 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 dependency graph for the running example is shown in Fig. 5, assuming the gray state as the start state. The throughput of an SDFG is limited by an infinite sequence of causal dependencies between the actor firings, captured by a causal dependency cycle in the dependency graph.

DEFINITION 12. (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 which model storage space are of interest to us as adding more tokens to these channels (i.e. increasing the storage space of the corresponding channel) may resolve causal dependency cycles and increase throughput.

DEFINITION 13. (STORAGE DEPENDENCY) *Given an SDFG (A_δ, C_δ) incorporating a storage distribution δ and its dependency graph Δ. A channel $c \in C_\delta$ has a storage dependency in Δ if and only if there exists a causal dependency in some dependency cycle of Δ via channel c_δ.*

8. DESIGN-SPACE EXPLORATION

Sec. 6 presents a technique to find the throughput for a given storage distribution. Using this technique, it is possible to find the trade-offs between the distribution size and the throughput, i.e., the *Pareto space*. Fig. 6 shows this Pareto space for our example. It shows that storage distribution $\langle 4, 2\rangle$ is the smallest distribution with a throughput for actor c larger than zero. The throughput of c can never go above 0.25, as actor b always has to fire twice (requiring 4 time steps) for one firing of c. With a distribution size of 10 tokens (or more), the maximal throughput can be achieved.

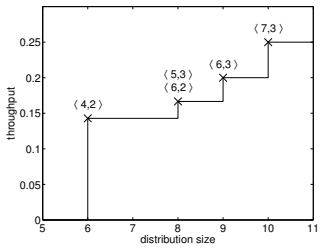

Figure 6: Pareto space for our example.

DEFINITION 14. (MINIMAL STORAGE DISTRIBUTION) *A storage distribution δ with throughput τ is minimal if and only if for any other storage distribution δ' with throughput τ', $|\delta'| < |\delta|$ implies $\tau' < \tau$ and $|\delta'| = |\delta|$ implies $\tau' \leq \tau$.*

Distributions $\langle 4,2\rangle$, $\langle 5,3\rangle$, $\langle 6,2\rangle$, $\langle 6,3\rangle$ and $\langle 7,3\rangle$ in our example are minimal, but distribution $\langle 5,2\rangle$ is not.

Algorithm 1 is used to find all minimal storage distributions for an SDFG. The algorithm starts the design-space exploration from a distribution δ. It computes the storage dependency graph Δ and throughput τ for this distribution. The distribution-throughput pairs are kept in a set D. The algorithm continues by recursively increasing the storage space of each channel c which has a storage dependency in Δ. The storage space of c is increased by the step size of the channel as explained in Sec. 4. All Pareto points for an SDFG are found using the following theorem.

THEOREM 2. *The set of all storage distributions contained in D which is constructed using Algorithm 1 starting from $D = \emptyset$ and the storage distribution $\delta_0 = \langle 0,\ldots,0\rangle$ contains all minimal storage distributions.*

The theorem shows that set D contains all minimal storage distributions. However, it may also contain non-minimal distributions. Sorting all distributions in D according to their throughput and a single traversal through the sorted list is sufficient to extract all Pareto points from D.

In practice it is not necessary to start from the distribution $\langle 0,\ldots,0\rangle$. For each channel, a lower bound on the storage space required to avoid deadlock (i.e. throughput equal to zero) can be computed [1, 12].

Algorithm 1 Find all minimal storage distributions

Input: An SDFG G with a storage distribution δ
Result: A set D of pairs (storage distribution, throughput)

```
1: procedure findMinStorageDist(G, δ)
2:     Create SDFG G_δ which models δ in G
3:     Compute throughput τ and dependency graph Δ of G_δ
4:     D ← D ⋃ (δ, τ)
5:     Let S be the set of storage dependencies in Δ
6:     for each channel c in S do
7:         δ_n ← δ
8:         δ_n(c) ← δ(c) + step(c)
9:         D ← D ⋃ findMinStorageDist(G, δ_n)
```

9. ABSTRACT DEPENDENCY GRAPH

The number of nodes in a dependency graph can be large as it contains a node for each actor firing which occurs during the periodic phase. As a result, cycle detection can be very time consuming. An abstract version of the dependency graph can be constructed in which the number of nodes is equal to the number of actors in the SDFG, which enables faster cycle detection.

DEFINITION 15. (ABSTRACT DEPENDENCY GRAPH) *Given an SDFG (A_δ, C_δ) incorporating a storage distribution δ and its dependency graph (D, E). The abstract dependency graph (D_a, E_a) contains an abstract dependency node $d_a \in D_a$ for each actor $a \in A_\delta$. Each dependency edge $(a_k, b_l) \in E$ has a corresponding edge (d_a, d_b) in E_a.*

In practice, the abstract dependency graph can be constructed by traversing through the cycle in the state-space once. An important property of the abstract dependency graph is that it maintains all storage dependencies present in the dependency graph. So, Algorithm 1 can still be used to find all minimal storage distributions.

THEOREM 3. *The set of storage dependencies of an abstract dependency graph contains all storage dependencies of the corresponding dependency graph.*

10. IMPLEMENTATION

We developed a tool, called *SDF*^{3} [20], that implements among other things the techniques presented in this paper. It takes an XML description of an SDFG as input and adds channels to model storage space. The result is C++ code of a program which performs the actual design-space exploration. This program follows the structure of Algorithm 1 starting from the storage distribution δ given by the lower bounds for storage space of the channels. It sets the initial state of the SDFG according to storage distribution δ. Using the technique described in Sec. 6, the throughput is computed, and as a byproduct the abstract dependency graph. A cycle detection algorithms is then used to find all storage dependencies. The algorithm recursively enlarges all channels with a storage dependency. During this process, an increase of some channels, in different orders, can result in the same storage distribution. To avoid that such storage distributions and their extensions are explored more than once, a dynamic programming approach is used. Instead of the set D, the program maintains a sorted list of all storage distributions it has seen. This increases the speed of the dynamic programming and it allows simple extraction of the minimal storage distributions from the list.

11. EXPERIMENTAL RESULTS

We have performed experiments to see how our approach performs in practice on a number of real DSP and multimedia applications modeled as SDFGs. From the DSP domain, the set contains a modem [5], a satellite receiver [16] and a sample-rate converter [5], and from the multimedia domain an MP3 decoder and an H.263 decoder. We also used our own example SDFG shown in Fig. 2 and the often used bipartite SDFG from [5]. For each of the SDFGs, 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 are summarized in Tab. 1. It shows the number of actors and channels in each graph and 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. It also shows the number of Pareto points and the number of minimal storage distributions that were found during the design-space exploration. The results show that one Pareto point of our example graph contains two different minimal storage distributions. In all other situations, each Pareto point contains a single storage distribution.

An upper bound on the storage space required for each channel to achieve maximal throughput with finite channel capacities can be found [11]. This upper bound and the lower bound explained in Sec. 8, can be used to compute the number of different storage distributions in the design-space (see row '#Distr. in space' of Tab. 1). The next row shows

Table 1: Experimental results.

	Example	Bipartite	Sample Rate	Modem	Satellite	MP3	H.263	H.263 (q: 3)	H.263 (q: 9)	H.263 (q: 27)
#actors/ #channels	3/2	4/4	6/5	16/19	22/26	13/12	4/3	4/3	4/3	4/3
Min. throughput > 0	1/7	0.04	0.15	0.03	0.18	$7{\cdot}10^{-3}$	$5{\cdot}10^{-5}$	$5{\cdot}10^{-5}$	$5{\cdot}10^{-5}$	$5{\cdot}10^{-5}$
Distr. size	6	28	32	38	1542	12	4753	4753	4753	4753
Max. throughput	1/4	0.06	0.17	0.06	0.23	$8{\cdot}10^{-3}$	$1{\cdot}10^{-4}$	$1{\cdot}10^{-4}$	$1{\cdot}10^{-4}$	$1{\cdot}10^{-4}$
Distr. size	10	35	34	40	1544	16	8006	8006	8012	8021
#Pareto points	4	8	3	3	2	3	3254	1086	364	123
#Min. distr.	5	8	3	3	2	3	3254	1086	364	123
#Distr. in space	27	$1{\cdot}10^{8}$	$9{\cdot}10^{12}$	$1{\cdot}10^{10}$	$2{\cdot}10^{65}$	4096	$3{\cdot}10^{10}$	$3{\cdot}10^{10}$	$3{\cdot}10^{10}$	$3{\cdot}10^{10}$
#Distr. checked	7	51	3	4	4	7	$292{\cdot}10^{3}$	28720	3613	558
Overestimation	-	-	-	-	-	-	-	0%	$3{\cdot}10^{-4}$%	$2{\cdot}10^{-3}$%
Max. #states visited	21	652	$6{\cdot}10^{6}$	134	10377	33579	$8{\cdot}10^{6}$	$8{\cdot}10^{6}$	$3{\cdot}10^{6}$	905657
Max. #states stored	2	20	5328	2	241	212	1124	375	125	43
Exec. time	$1ms$	$1ms$	$1ms$	$2ms$	$7ms$	$2ms$	$53min$	$5min$	$36ms$	$7ms$

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. This shows that the algorithm successfully prunes the design-space.

The algorithm computes the throughput for each storage distribution it tries. This is done via a self-timed execution of the SDFG (see 'Max. #states visited' of Tab. 1) in which a selected number of states must be stored (see Sec. 6 and 'Max. #states stored' of Tab. 1).

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. 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. By quantizing the size with which the storage space is increased (line 8 of Algorithm 1), the number of Pareto points can be limited. Several experiments have been performed in which the step size of the channels in the H.263 decoder was multiplied with a quantization factor of 3, 9 and 27. The results for these experiments are shown in the last three columns of Tab. 1. It shows that the quantization drastically improves the run time of the design-space exploration at the cost of a reduced number of Pareto points found. Quantization of the step size can also lead to overestimation of the required storage space for a given throughput. The experiments show that the overestimation for the H.263 decoder is very small (see row 'Overestimation'). In fact, only the storage distribution which achieves maximal throughput is overestimated.

From the experiments, we conclude that it is feasible to perform a design-space exploration for reasonable application kernels.

12. CONCLUSION

We have presented a method to explore the trade-offs between the throughput and memory usage for SDFGs. It differs from existing buffer sizing methods as it can determine exact minimum memory bounds for any achievable throughput of the graph. Other methods can only determine an upper bound on the minimal memory requirement for the lowest or highest throughput of the graph. The current experiments show that, despite the complexity of the problem, it is possible to perform a design-space exploration for real application kernels. If run times nevertheless become a problem, our techniques can be combined with approximation or heuristic techniques to prune the design-space. A simple example of such an approximation algorithm was implemented and tested on an H.263 decoder with good results. Our objective is to integrate this work in a predictable design flow for systems-on-chip based on SDFGs in the style of [15].

13. REFERENCES

[1] M. Adé et al. Data minimisation for synchronous data flow graphs emulated on dsp-fpga targets. In *DAC'97, Proc.* (1997), ACM, p. 64–69.

[2] K. Altisen et al. A methodology for the construction of scheduled systems. In *Int. Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systems, Proc.* (2000), Springer-Verlag, p. 106–120.

[3] T. Amnell et al. Times: a tool for schedulability analysis and code generation of real-time systems. In *FORMATS'03, number 2791 in LNCS* (2004), Springer-Verlag, p. 60–72.

[4] S. Bhattacharyya et al. *Software Synthesis from Dataflow Graphs*. Kluwer, 1996.

[5] S. Bhattacharyya et al. Synthesis of embedded software from synchronous dataflow specifications. *Journal on VLSI Signal Process. Syst. 21*, 2 (1999), p. 151–166.

[6] J. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory using the Token Flow Model.* PhD thesis, University of California, Berkeley, CA, 1993.

[7] M. Geilen, T. Basten, and S. Stuijk. Minimising buffer requirements of synchronous dataflow graphs with model-checking. In *DAC'05, Proc.* (2005), ACM, p. 819–824.

[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 *ACSD'06, Proc.* (2006), IEEE.

[9] R. Govindarajan et al. Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks. *Journal of VLSI Signal Processing 31*, 3 (2002), p. 207–229.

[10] C.-T. Hwang et al. A formal approach to the scheduling problem in high-level synthesis. *IEEE Trans. on Computer-Aided Design 10*, 4 (1991), p. 464–475.

[11] E. Lee and D. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. on Comp. 36*, 1 (1987), p. 24–35.

[12] P. Murthy and S. Bhattacharyy. Shared memory implementations of synchronous dataflow specifications. In *DATE'00, Proc.* (March 2000), IEEE, p. 404–410.

[13] Q. Ning and G. Gao. A novel framework of register allocation for software pipelining. In *Symp. on Principles of Programming Languages, Proc.* (1993), ACM, p. 29–42.

[14] H. Oh and S. Ha. Efficient code synthesis from extended dataflow graphs. In *DAC'02, Proc.* (2002), IEEE, p. 275–280.

[15] 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, Proc.* (2003), ACM, p. 63–72.

[16] S. Ritz et al. Scheduling for optimum data memory compaction in block diagram oriented software synthesis. In *Int. Conf. on Acoustics, Speech, and Signal Processing, Proc.* (1995), IEEE, p. 2651–2654.

[17] 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, Proc.* (2001), IEEE, p. 53–57.

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

[19] S. Stuijk, M. Geilen, and T. Basten. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. *Tech. Report, TU Eindhoven.* http://www.es.ele.tue.nl/esreports/esr-2006-01.pdf

[20] S. Stuijk, M. Geilen, and T. Basten. SDF^3: SDF for free. In *ACSD'06, Proc.* (2006), IEEE. http://www.es.ele.tue.nl/sdf3