technische universiteit eindhoven

Exploring Trade-Offs in Buffer Requirements and Throughput Constraints for Synchronous Dataflow Graphs

Sander Stuijk, Marc Geilen, Twan Basten

ES Reports
ISSN 1574-9517

ESR-2005-07
14 July 2005

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems

http://www.es.ele.tue.nl/esreports
esreports@es.ele.tue.nl

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems
PO Box 513
NL-5600 MB Eindhoven
The Netherlands

Exploring Trade Offs in Buffer Requirements and Throughput Constraints for Synchronous Dataflow Graphs

Sander Stuijk, Marc Geilen and Twan Basten
Eindhoven University of Technology, P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands.
{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 an exact method to determine the minimal storage space needed to execute a graph under a given throughput constraint. We also show how this method can be used to chart the Pareto space of throughput and storage trade-offs. The feasibility of the approach is demonstrated with a number of examples.

1 Introduction

Consumers have high expectations about the quality delivered by new multimedia products such as PDAs, settop-boxes, etcetera. One aspect that plays an important role in the perceived quality is timing. For instance, in a modern television, a new frame should be displayed each 1/100th of a second. The number of times this deadline is missed should be minimized. However, designing a system with this constraint in mind is getting more complex. One reason for this is the growing complexity of hardware, because of the integration of many components in *multi-processor systems-on-chip*. Another reason is the eminent trend to execute many applications and (sub-)tasks concurrently in these systems. As a result, an enormous amount of use cases must be checked. For example, in a modern television platform 60 applications are running in parallel, corresponding to an order of 60! possible use cases. It is impossible to verify all these cases through testing and simulation. This has motivated researchers to emphasize the ability to analyze and predict the behavior of applications and platforms without extensive simulations. Real-time requirements in media systems have put the main focus on predicting the *timing* behavior of complex media systems. This development has accelerated the use of *models of computation* as a class of them allows analysis of the system at design time. An example of such a model of computation are Synchronous Dataflow Graphs (SDF) [LM87]. They are often used to specify compute intensive kernels. The nodes in an SDF graph 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 SDF graphs has been studied before by several authors. See for instance [ALP97, BML96, BML99, GBS05, GGD02, HLH91, MB00, NG93, OH02]. 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 [ALP97, BML96, BML99, GBS05, MB00, OH02] may result in an implementation of the SDF graph that cannot be executed within these timing constraints. It may be necessary to take the timing constraint into account while minimizing the buffers. Several approaches have been proposed for minimizing buffer requirements under a throughput constraint. In [GGD02], a technique based on linear programming is proposed to calculate a schedule that realizes the maximal throughput while it tries to minimize the buffer sizes. Hwang et al. propose a heuristic that can take resource constraints into account [HLH91]. This method is targeted towards a-cyclic graphs and it always maximizes the throughput rather than using a throughput constraint. Thus, it could lead to additional resource requirements. In [NG93], buffer minimization for maximal throughput of a subclass of SDF graphs (homogeneous) is studied. An integer linear programing formulation is presented that minimizes the buffer requirements. In general, the minimal buffer sizes obtained with this approach cannot be translated to exact minimal buffer sizes for arbitrary SDF graphs. The existing approaches generate schedules that realize a maximal throughput and a buffer size as close as possible to the minimal size; none of the techniques is exact. We propose, in contrast with existing work, an *exact* technique to determine the *trade-offs* (*Pareto points*) between the throughput and minimal storage requirements for an SDF graph (including the schedules realizing these trade-offs).

The buffer minimization problem is known to be NP-complete [BML96]. Other researchers have successfully applied explicit state-space exploration techniques to solve NP-complete (and even worse) scheduling problems [AGS00, AFM$^+$04, SG01]. Because of the promising results of these approaches, we decided to develop a method based on explicit state-space exploration for the buffer minimization problem. In [GBS05], we present an approach to determine the exact minimal buffer requirements to execute an SDF graph with a deadlock-free schedule. No timing constraints were taken into account. In this paper, we extend our existing approach to buffer sizing under a throughput constraint.

We propose a technique for finding the smallest buffer size for which a given SDF graph can be executed with a schedule that meets a given throughput constraint. We first introduce the timed SDF model and formalize the storage requirements and scheduling of an SDF graph. Subsequently, we motivate our work on minimizing buffer requirements under arbitrary throughput constraints using a simple example. Next, we show how the problem can be modeled as a state-space exploration problem. A tool, based on model-checking techniques, to perform this exploration is presented. We show that our technique can be used to perform a design-space exploration to find the trade-offs between throughput and memory consumption of an SDF graph. The feasibility of the approach is demonstrated with a number of case studies.

2 Synchronous Dataflow Graphs and Time

An example of a Synchronous Dataflow Graph (SDF) is depicted in Fig. 1. The nodes of an SDF graph 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 SDF graphs 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 *rates* of the ports, and are visualized as port annotations. The edges in the graph, called *channels*, represent data that is communicated from one actor to another. The channels may contain *initial tokens* present at start time. Formally, an SDF graph is a pair (A, C) consisting of the set A of actors and the set C of channels.

The time needed for one firing of an actor is called its *execution time*, measured in discrete

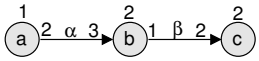

Figure 1: Example SDF graph.

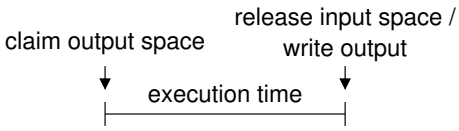

Figure 2: Timed firing of an actor.

time steps. It is a mapping from A to the natural numbers $\mathbb{N}$. The execution times of actors are depicted above the nodes in the graph. The timed firing of an actor is modeled as follows (see Fig. 2). At the start of the firing, the actor claims space in the channels for which it is producing data; this represents the allocation of storage space for writing the results of the computation. At the end of the firing, the actor writes output and releases the space of the input tokens; this represents the consumption of input data that is no longer needed. The chosen abstraction is conservative with respect to storage and throughput if in a real implementation space is claimed later, or released earlier or tokens are written earlier. The firing of an actor can start as soon as enough tokens are available on the input, enough space is present on the output, and the previous firing of the actor has finished. The latter implies that no auto-concurrency is possible. This is a logical assumption in the context of embedded systems in which an actor is usually mapped to a single processor which does not support concurrent execution of code.

3 Storage Requirements

To allow actors to execute asynchronously, some storage space is required for the channels to store data that is produced by one actor, but has not yet been consumed by the next. If the storage space in a channel is insufficient, an actor may not be able to fire. The firing of the actor must then be delayed till enough storage space is available. This may reduce the throughput. The situation in which this storage space never becomes available is called a deadlock situation. The graph can then no longer perform its computation. The class of *consistent* SDF graphs [Lee91] is the class of graphs that allow a deadlock-free execution within bounded memory. Since consistency is straightforward to check, and since the buffer sizing problem is trivial for inconsistent graphs, we focus on consistent graphs.

The 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 the SDF graph 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 SDF graphs with minimal storage space [MB00]. 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 multi-processor systems, as memories are not always shared between all processors.

Table 1: Schedule for motivating example with storage distribution $\langle \alpha, \beta \rangle \mapsto \langle 4, 2 \rangle$.

Time step:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	a	a	b	_b	a	b	_b	c	_c							
								a	a	b	_b	a	b	_b	c	_c
															a	a

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 [ALP97, BML99]. Hybrid forms of both options can be used [GBS05].

In this paper, we deal with the situation in which channels cannot share memory space. This gives a conservative bound on the required memory space when the SDF graph is implemented in a real system. When a system supports shared memory, the SDF graph may require less memory, but it will never require more memory than determined by our method.

The maximum number of tokens stored in each channel (*channel capacity*) during the execution of an SDF graph is called a *storage distribution*.

Definition 1. (Storage distribution) *A storage distribution of an SDF graph (A, C) is a mapping $\gamma : C \to \mathbb{N}$, that associates with every channel $c \in C$, the capacity of the channel.*

A possible storage distribution for the SDF graph shown in Fig. 1 would be $\gamma(\alpha) = 4$ and $\gamma(\beta) = 2$, denoted as $\langle \alpha, \beta \rangle \mapsto \langle 4, 2 \rangle$. The storage space required for a storage distribution is called the *distribution size*.

Definition 2. (Distribution size) *The size of a storage distribution γ is given by:*

$$sz(\gamma) = \sum_{c \in C} \gamma(c).$$

The distribution size for our example is 6 tokens. A storage distribution γ_1 is said to be *smaller* than a storage distribution γ_2 if $sz(\gamma_1) < sz(\gamma_2)$.

4 Scheduling

In Sec. 2, we discussed the timed firing model for actors in an SDF graph. We can define a schedule of the graph as follows.

Definition 3. (Schedule) *A schedule σ for an SDF graph (A, C) is a function $A \times \mathbb{N} \to \mathbb{N}$ which maps each firing of each actor in A to a time instance. That is, $\sigma(a, i)$ represents the time at which the i-th firing of actor a starts. Schedule σ must satisfy the constraints that a firing starts as soon as enough input tokens are present, enough output space is available, and the previous firing of the actor has finished.*

A schedule for the SDF graph shown in Fig. 1 with the storage distribution $\langle \alpha, \beta \rangle \mapsto \langle 4, 2 \rangle$ is shown in Table 1. The actors in a single column fire concurrently. The concurrent firing can be realized, as we are targeting multi-processor systems. The symbol $\_a$ is used to indicate that the execution of actor a is continuing from the previous time step. The schedule from time step 3 to time step 9 is repeated indefinitely. A new iteration is initiated after every 7 time steps. A schedule that contains such a repetitive pattern is called a *periodic schedule*. If the periodic schedule is executing the repetitive pattern, it is said to be in the *periodic phase*. This periodic phase is never left. All schedules of consistent graphs contain such a periodic phase, as is shown in Sec. 6. In our example, we enter this periodic phase in time step 3. If the schedule is not yet in the periodic phase, then we say that the schedule is in the *transient phase*. Time steps 1 and 2 in our example belong to this transient phase.

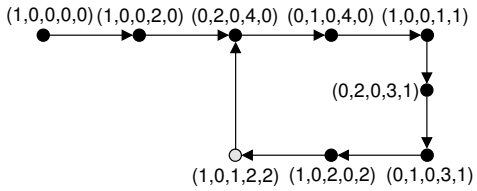

Figure 3: SDF state space of the example SDF graph.

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. The fixed rates of the actors ensure that the number of times actors fire with respect to each other (repetition vector [Buc93]) is constant. In other words, the throughput of each pair of actors in a graph is related to each other via a constant. The throughput is defined as follows.

Definition 4. (THROUGHPUT) *The throughput of an actor a with schedule σ is equal to the average number of firings of a per time step.*

Consider again our example: actor c fires for the first time at time step 8. At that moment, the actor is in the periodic phase of the schedule and fires each 7 time steps. In streaming applications, 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. Our example has always 7 time steps between two firings of c in the periodic phase. So, the throughput of c is $1/7$.

It is interesting to consider how a schedule with maximal throughput for a given storage distribution can be constructed. An actor can fire as soon as its input data is available, there is space on the output channel, and its previous firing has finished. Free space available in the output channel cannot be used by another actor to increase the throughput. So, throughput cannot be increased by delaying the firing. An optimal schedule is therefore obtained by firing all actors as soon as they are enabled.

6 Timed SDF State Space

The observation that every actor must be fired as soon as possible to achieve maximal throughput implies that the firing of actors is deterministic. Hence, the execution of the entire SDF graph is deterministic for any given storage distribution. In other words, there exists exactly one throughput-optimal schedule for the SDF graph with the given storage distribution. The schedule maps actor firings to a time instance. At each time instance, the SDF graph is fully characterized by the following notion of a state.

Definition 5. (SDF STATE) *The state q of an SDF graph (A, C) is the $n+m$ tuple $(t_1, \ldots, t_n, s_1, \ldots, s_m)$ with $n = |A|$ and $m = |C|$, t_i the time remaining for the completion of the firing of actor $a_i \in A$ or zero if a_i is not firing, and s_i the number of tokens stored in channel $c_i \in C$.*

Assume that the graph is in a state q_1. When the schedule advances to the next time instance, the time remaining for the completion of each actor firing is lowered till it reaches zero, and the corresponding token consumptions and productions are performed (as illustrated in Fig. 2). Furthermore, each actor $a \in A$ that can be fired is fired, which means that the corresponding t_i in the state is set to the execution time of a. Consider again our example SDF graph shown in Fig. 1 with storage distribution $\langle 4, 2 \rangle$. Initially, no tokens are stored in the channels. Actor a is enabled and thus immediately fires which means that the initial state becomes $(1, 0, 0, 0, 0)$.

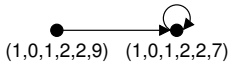

Figure 4: Reduced state space of the example SDF graph.

After 1 time unit, a has finished its firing, it produced 2 tokens on channel α, and a new firing of a has started as there is enough space available in α. The state of the SDF graph is thus equal to $(1, 0, 0, 2, 0)$. At the next time unit, the firing of a finishes, it produces 2 tokens in channel α, and b can thus be fired, the state is then $(0, 2, 0, 4, 0)$. Following the schedule of the graph as shown in Tab. 1, the traversed states are shown in Fig. 3. Such a traversal through the state space can be constructed for any SDF graph (A, C) and storage distribution γ. In each transition the SDF graph goes from a state q_1 to a state q_2. The transition from q_1 to q_2 is deterministic as the actor firings are deterministic.

Theorem 1. *Given a consistent SDF graph and assuming that actors fire as soon as they are enabled, the timed behavior of the graph is either periodic or it deadlocks (meaning that at some point in time no actors are enabled anymore).*

Proof. The execution time of each actor a is finite and the number of tokens which can be stored in each channel $c \in C$ is also finite. So, there exists only a finite number of different states for a given SDF. Assuming absence of deadlock, the execution of a consistent SDF graph is infinitely long. According to the pigeon hole principle, at least one state is visited infinitely often. Since the behavior is furthermore deterministic, it must be periodic. □

Observe that a deadlock results in a self-loop transition from a state to itself (i.e., a cycle). The $t_1, \ldots, t_n$ components of such a state are all 0, indicating that no actors are firing, and the $s_1, \ldots, s_m$ components are all such that no actor is enabled (due to lack of input tokens or output space). The following property follows immediately from Theorem 1 and this observation.

Property 1. *The state space of a consistent SDF graph contains always exactly one cycle.*

7 Throughput Calculation

In the situation that the SDF graph is not in a deadlock, the cycle in an SDF state space contains precisely all states visited in the periodic phase of the schedule. The throughput of an actor a can easily be determined from this cycle. Each state in the cycle represents one time instant. The throughput is then equal to the number of firings of a in the cycle divided by the number of states in the cycle. As explained, the transient phase of the behavior can be ignored.

Property 2. *The throughput of an actor a in a consistent SDF graph is either 0 (in case of deadlock) or it equals the number of firings of a in the cycle of the state space divided by the duration of the cycle.*

In order to calculate the throughput of an actor a, only the number of firings of a on the cycle and the number of states on the cycle, i.e., the duration, are needed. To detect the cycle, only the states which represent the firing of a must be kept. The other states can be abstracted away in an additional dimension, d_a, of the state, which represents the number of time instances between the last two consecutive firings of the actor a. In terms of the SDF state space, this is the number of states between the last two consecutive firings of a. The SDF state space shown in Fig. 3 for example can be reduced to the state space shown in Fig. 4 when interested in the throughput of c. Only the gray state in Fig. 3 is kept. The first state in the reduced state space is reached when c fires for the first time, which is 9 time instances after the start of the execution of the SDF graph. Next, the cycle in the state space shown in Fig. 3 is traversed

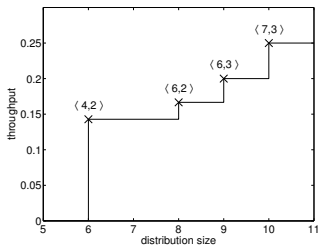

Figure 5: Pareto space for our example.

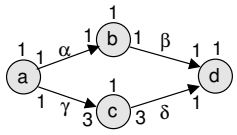

Figure 6: Channel capacities above lower bound.

in 7 time instances. In the reduced state space this is represented by a transition from state $(1, 0, 1, 2, 2, 9)$ to state $(1, 0, 1, 2, 2, 7)$. After this, the cycle in the state space is traversed each 7 time instances. This is represented by the cycle from state $(1, 0, 1, 2, 2, 7)$ to itself.
It is clear that the reduction of the state space preserves all information necessary to calculate the throughput of an actor. However, it is much more efficient in terms of memory and execution time to construct the reduced state space than it is to explicitly construct and store the entire timed state space. The throughput of an actor a is now given by the sum of the total amount of time between all firings of a on the cycle (sum of d_a for all states on the cycle) divided by the number of firings of a on the cycle (number of states on the cycle). For our example graph, there is 1 state on the cycle with d_a equal to 7. So, the throughput is 1/7. Observe that during construction of the reduced state-space, it is straightforward to detect deadlock and to construct the schedule that yields the computed throughput.

8 Storage-Throughput Design Space

So far, we have seen how to compute the throughput for a given storage distribution. In general, we are interested in storage throughput trade-offs.

The schedule of Table 1 for our simple example (Fig. 1) is based on storage distribution $\langle \alpha, \beta \rangle \mapsto \langle 4, 2 \rangle$. In this distribution, the actors a and b and the actors b and c cannot fire concurrently. Increasing the channel capacities allows for the pipelined concurrent execution of these actors. For example, if the channel capacity of α is increased to 6 tokens, then the throughput increases to 1/6. Figure 5 shows the trade-offs, i.e., the *Pareto space*, between the distribution size and the throughput. The figure shows that the storage distribution $\langle 4, 2 \rangle$ is the smallest storage distribution which has a throughput larger than zero. The throughput of the actor c in the graph can never go above 0.25, as actor b always has to fire twice (requiring 4 time steps) for one firing of c. So, with a distribution size of 10 tokens, the maximal throughput can be achieved. Further increasing the distribution size will never lead to an increase in the throughput.

A storage distribution for which there exists no other storage distribution that is smaller but has an equal throughput is called a *minimal storage distribution*. The distributions $\langle 4, 2 \rangle$ and $\langle 6, 2 \rangle$ in our example are such minimal storage distributions, but distribution $\langle 5, 2 \rangle$ is not. Multiple storage distributions with the same distribution size and throughput may exist. Consider the graph shown in Fig. 6. Storage distributions $\langle \alpha, \beta, \gamma, \delta \rangle \mapsto \langle 1, 2, 3, 3 \rangle$ and $\langle 2, 1, 3, 3 \rangle$ are both

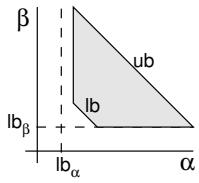

Figure 7: Design-space boundaries.

minimal and realize the same throughput for actor d. So, minimal storage distributions for a certain throughput are not unique. Different distributions of the same size may be interesting if the memory onto which the channels are mapped is distributed. This aspect is not further considered in this paper, but it can be taken into account straightforwardly as extra constraints on the channel capacities.

From [ALP97, Mur96], we know how to determine a lower bound on the channel capacity of a channel with a given production rate, consumption rate and initial number of tokens for a positive (non-zero) throughput. In [GGD02], a method is described to compute an upper bound on the storage space for all the combined channels in an SDF graph for the highest possible throughput. Using [GBS05], we can determine a lower bound for all the combined channels. These lower bounds and upper bound limit the design space that must be explored to find a minimal storage distribution for arbitrary throughput. Figure 7 illustrates this for an SDF graph with two channels α and β with individual lower bounds lb_α and lb_β and combined lower and upper bounds lb and ub. The gray area contains all minimal storage distributions for any positive throughput for the graph. Note that the lower bounds for the channels do not always give the minimal storage distribution for the lowest positive throughput. A larger channel capacity for one or more of the channels in the graph may be required for this. For instance, either channel α or β in the SDF graph of Fig. 6 must be above the lower bound of 1 to realize a positive throughput (i.e. to not deadlock). To find a minimal storage distribution for a given throughput constraint, only storage distributions within these bounds must be considered. Despite this restriction of the design space, the number of distributions remaining is still exponential in the number of channels and the difference between the minimal (lower bound) and maximal distribution size (upper bound).

9 Design-Space Exploration

In Sec. 7, a technique has been presented to find the throughput for a given storage distribution. Using this technique, it is possible to find the Pareto space of throughput and storage trade-offs. As mentioned, an example of such a space is shown in Fig. 5. All points to the right of the curve are *feasible* solutions and all points to the left are *infeasible* solutions. The minimal storage distributions are called *Pareto points*. Exploration of the design space in order to find all Pareto points is done as follows.

We have to search the two dimensions of the design space. To search the distribution-size dimension, we apply a divide-and-conquer strategy. In the throughput dimension, we apply a binary search to determine the maximal achievable throughput given a distribution size. An important observation is that throughput is monotonic in the distribution size, i.e. with increasing distribution size, the throughput will not decrease. This has an interesting consequence. If, for example, the maximal throughput of an SDF graph is the same for distribution sizes 3 and 6, the throughput for distribution sizes 4 and 5 must also be the same. Thus, we try to find these intervals in the distribution-size dimension as this minimizes the number of points of the design space to be searched. Using the techniques of [GBS05, GGD02], we first determine

a lower and upper bound on the meaningful distribution-size interval (the lb and ub values of Fig. 7), and compute the maximum throughput for these points. For the lower bound, we compute the maximum throughput with the binary-search method explained below. The maximal achievable throughput for the upper bound equals the maximal achievable throughput of the SDF graph, which can be computed, e.g., via the technique of [GG93]. Then, we compute in a binary search the achievable throughput for the distribution size halfway the interval recursively till the point that the maximal throughput for the minimum and maximum values of an interval is the same. This results in an efficient and complete search of the distribution-size dimension of our design space. The second problem is to determine the maximal achievable throughput given a specific distribution size, i.e., to search the throughput dimension of the design space. We apply a standard binary search. The maximal throughput of an SDF graph can serve as the upper bound for the search. An obvious throughput lower bound is zero. We can further optimize the search process. The above mentioned monotonicity property implies that the maximal throughput found for a specific distribution size can serve as a lower bound for the binary search for maximal achievable throughput for a larger distribution size. Thus, if we organize the traversal of the distribution-size dimension in such a way that we build up the Pareto curve from left to right, we can incrementally improve the throughput lower bound for consecutive binary searches in the throughput dimension.
To realize the above mentioned binary search, it remains to decide whether a given throughput can be obtained with a given distribution size. To answer this question, all possible storage distributions of the given size must be searched till one is found of which the maximum throughput is larger or equal to the desired throughput. Each storage distribution can be checked by looking at the length of the cycle in the state space, as explained in Sec. 7. If no storage distribution is found with a throughput larger or equal to the desired throughput, the binary search is continued in the bottom half of the throughput interval. In case a storage distribution is found which realizes the desired throughput, the binary search continues in the top part of the throughput interval. The throughput lower bound is then set to the throughput found so far.

To conclude, the complexity of the algorithm to search the storage-throughput design space is exponential as both the state space of an SDF with a single storage distribution and the number of different storage distributions per distribution size may be exponentially large. Our experiments presented in Sec. 11 show that despite this complexity it is possible to search the design space for realistic problems.

10 Implementation

As in [GBS05], we first implemented the SDF state space and its exploration in the general-purpose model-checker SPIN [Hol04]. However, SPIN does not allow us to easily exploit all properties offered by the SDF model and its state-space, as discussed in Sec. 7. We either have to implement the entire design space in SPIN or we use SPIN only for exploring a single distribution-size throughput trade-off. The first option leads to an unmanageable state-space explosion because SPIN maintains all state spaces for all storage distributions simultaneously; the second option also suffers to some extent of this problem but in addition it leads to so many calls to SPIN that the start-up time of SPIN becomes a bottleneck. Therefore, we developed our own timed SDF model-checking tool, called *buffy*, that enables an automatic design-space exploration using the method described in the previous section. It takes an XML description of an SDF graph as input. If the user is interested in only part of the storage-throughput space, it is possible to set the maximum distribution size and bounds on the throughput. The program computes lower bounds on the channels which are required for a positive (non-zero) throughput. Next, it outputs the C++ code for a program which performs the design-space exploration. The remainder discusses the implementation of this program generated by buffy.

```
#define CH(c)                sdfState.ch[c]
#define CHECK_TOKENS(c,n)    (CH(c) >= n)
#define CHECK_SPACE(c,n)     (CH(c) <= sz[c] - n)
#define CONSUME(c,n)         CH(c) = CH(c) - n;
#define PRODUCE(c,n)         CH(c) = CH(c) + n;
#define ACT_CLK(a)           sdfState.act_clk[a]
#define LOWER_CLK(a)         if (ACT_CLK(a)>0) {ACT_CLK(a) = ACT_CLK(a)- 1;}

State sdfState;
short sz[2];

int execSDFgraph() {
  while (true) {
    LOWER_CLK(0); LOWER_CLK(1); LOWER_CLK(2);
    sdfState.dist = sdfState.dist + 1;

    if (ACT_CLK(0)==0 && CHECK_SPACE(0,2)) { ACT_CLK(0) = 1; }
    if (ACT_CLK(1)==0 && CHECK_TOKENS(0,3) && CHECK_SPACE(1,1)) { ACT_CLK(1) = 2; }
    if (ACT_CLK(2)==0 && CHECK_TOKENS(1,2)) { ACT_CLK(2) = 2; }

    if (ACT_CLK(0) == 1){ PRODUCE(0,2); }
    if (ACT_CLK(1) == 1){ CONSUME(0,3); PRODUCE(1,1); }
    if (ACT_CLK(2) == 1){ CONSUME(1,2);
          if (storeState(sdfState) != 0) return 1;
          sdfState.dist = 0;
    }
    /* deadlock detection omitted */
  }
}
```

Figure 8: C code generated by buffy for the example SDF graph.

The algorithms used for the binary search and the divide-and-conquer are straightforward and therefore not explained. The implementation of the execution of an SDF graph and the calculation of the throughput for the SDF graph with a given storage distribution is the interesting part. The program corresponding to the example shown in Fig. 1 is shown in Fig. 8. The maximal storage space allocated for each channel in the graph is encoded using the global variable `sz`. The array has two elements representing the two channels α (`sz[0]`) and β (`sz[1]`). The global variable `sdfState` contains the state of the SDF graph during its execution. The `State` is a structure which contains an array `act_clk` (actor clocks) with three elements representing the three actors a (`act_clk[0]`), b (`act_clk[1]`) and c (`act_clk[2]`). The value of each actor clock is equal to the time remaining for the completion of the firing of the actor when active, 0 otherwise. The `LOWER_CLK(n)` directive lowers the actor clock of n with one time step till it reaches zero. The structure contains also an array `ch` which gives the number of tokens stored in each channel and a variable `dist` to keep the number of time instances between two firings of the actor c of which we want to know the throughput.
The execution of the SDF graph for a given storage distribution is performed in the function `execSDFgraph`. Each iteration of the `while` loop specifies the execution of one time instance. The loop starts by advancing the time to the next time step. All actor clocks are lowered and the `dist` variable is increased. Next, it checks which actor can be fired. An actor can be fired, if it is currently not firing (i.e. its actor clock is zero) and there are sufficient tokens available on the input channels and there is enough space on the output channels. The latter two conditions are checked using the `CHECK_TOKENS(c,n)` and `CHECK_SPACE(c,n)` directive. These directive test respectively whether n tokens are available or n tokens can be stored in channel c. In the last part of the loop, it is checked which actors finish their firing in the current time instance (i.e. the remaining time is equal to 1). An actor whose firing finishes consumes the tokens from the input channels (`CONSUME`) and it produces tokens on the output channels (`PRODUCE`). The last actor to be checked is the actor whose throughput must be calculated (actor c in our case). If the firing of c finishes, the state must be added to the list of already visited states. This is

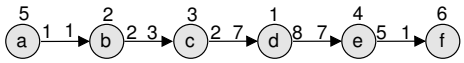

Figure 9: Sample rate converter ([BML99]).

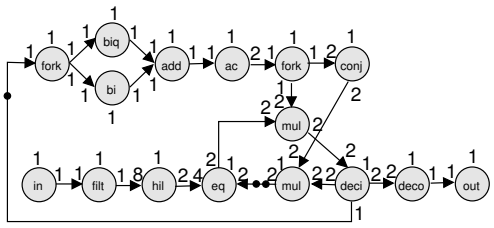

Figure 10: Modem application ([BML99]).

done using the `storeState` function. Using a hash table, it is quickly checked whether the state has been visited before. The execution of the SDF graph is continued until a cycle with positive throughput or a deadlock is detected. On a deadlock, the `while` loop ends and a throughput of 0 is returned. Details of the deadlock detection are omitted (no actors firing or enabled). If a cycle with positive throughput is detected, the execution of the `while` loop ends and the throughput of the actor is returned (in another function, not shown). If the explored graph and storage distribution form a Pareto point, a schedule is generated (not shown).

11 Experimental Results

We have performed a few experiments to see how our approach performs in practice. We have used the simple example of Fig. 1, the example SDF graphs of [BML99], depicted in Figures 9, 10 and 11 and a model of an H.263 decoder shown in Fig. 12. All experiments have been performed on a 800MHz Pentium III PC.

For each of the examples, 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. Fig. 13 shows, for example, the Pareto space of the modem. The results of all experiments are summarized in Tab. 2. The table 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 is shown in the row 'maximal throughput'. The distribution size needed to realize this maximal throughput is shown in the row below it. The next row shows the number of Pareto points that were found during the design-space exploration. During the exploration, the visited states (of the reduced state space) must be stored. The table shows the maximum number of states which must be stored in any single state space searched during the exploration. Since, the state spaces are very small, the execution time for the complete design-space exploration is acceptable for all graphs except the H.263 decoder. Its execution time is large due to the large number of Pareto points contained in the design 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 throughputs that are searched in the binary search, the number of Pareto points can be limited. This drastically improves the execution time of the design-

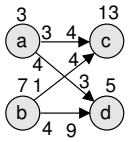

Figure 11: Bipartite graph ([BML99]).

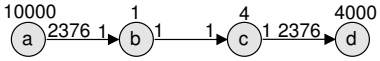

Figure 12: H.263 decoder.

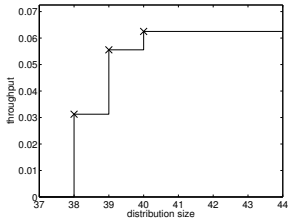

Figure 13: Pareto space for modem.

space exploration for the H.263 decoder. From the experiments, we conclude that it is feasible to perform a design-space exploration for reasonable application kernels. However, the design space may grow exponentially with the number of channels and the difference between the lower and upper bound on the distribution size (after all, the problem is NP-complete). An option to address this potential problem is to use heuristics that prune and limit the design-space and then search this restricted space using our techniques. In other words, the method presented in this paper can be used to refine the approximations of the design space given by existing heuristic methods.

12 Conclusions

In this paper, we have presented a method to explore the trade-off between the throughput and memory usage for SDF graphs. It differs from existing 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 realistic application kernels. We want to learn through more experiments how to further optimize and fine-tune this approach. However, increasing complexity of future applications may turn the exploration process into a time consuming process. We want to deal with this issue by combining our exploration strategy with (existing) heuristics that restrict the design-space to be searched. In this sense, our work is complementary to much of the existing work. Our ultimate objective is to integrate this work in a predictable design flow for systems-on-chip based on SDF in the style of [PBB$^+$03]. We also want to generalize the approach to more general data flow models like Boolean Dataflow,

Table 2: Experimental results.

	Example	Bipartite	Samp.Rate	Modem	H.263
Number of actors	3	4	6	16	4
Number of channels	2	4	5	19	3
Minimal throughput > 0	1/7	1/26	1/7	1/32	$4.6 \cdot 10^{-5}$
Distribution size	6	28	32	38	4753
Maximal throughput	1/4	1/18	1/6	1/16	$10 \cdot 10^{-5}$
Distribution size	10	33	34	42	7128
#Pareto points	4	8	2	3	2377
Maximum #states	2	17	234	2	2
Exec. time	$60ms$	$93ms$	$13ms$	$16s$	$67h$

for which the general problem is undecidable, but that does not exclude that it might work for many problems in practice.

References

[AFM+04] 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 *FORMATS'03, number 2791 in LNCS*, pages 60–72. Springer-Verlag, 2004.

[AGS00] K. Altisen, G. Gößler, and J. Sifakis. A methodology for the construction of scheduled systems. In *Int. Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systems, Proc.*, pages 106–120. Springer-Verlag, 2000.

[ALP97] M. Adé, R. Lauwereins, and J.A. Peperstraete. Data minimisation for synchronous data flow graphs emulated on dsp-fpga targets. In *DAC'97, Proc.*, pages 64–69. ACM, 1997.

[BML96] S.S. Bhattacharyya, P.K. Murthy, and E.A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.

[BML99] S. Bhattacharyya, P. Murthy, and E.A. Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal on VLSI Signal Process. Syst.*, 21(2):151–166, 1999.

[Buc93] J.T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory using the Token Flow Model*. PhD thesis, University of California, Berkeley, CA, 1993.

[GBS05] M. Geilen, T. Basten, and S. Stuijk. Minimising buffer requirements of synchronous dataflow graphs with model-checking. In *DAC'05, Proc.* IEEE, 2005.

[GG93] R. Govindarajan and G. Gao. A novel framework for multi-rate scheduling in dsp applications. In *Int. Conf. on Application-Specific Array Processors, Proc.*, pages 77–88. IEEE, 1993.

[GGD02] R. Govindarajan, G.R. Gao, and P. Desai. Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks. *Journal of VLSI Signal Processing*, 31(3):207–229, 2002.

[HLH91] 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*, 10(4):464–475, 1991.

[Hol04] G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison Wesley, 2004.

[Lee91] E.A. Lee. Consistency in dataflow graphs. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):223–235, 1991.

[LM87] E.A. Lee and D.G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1):24–35, 1987.

[MB00] P.K. Murthy and S.S. Bhattacharyy. Shared memory implementations of synchronous dataflow specifications. In *DATE 2000, Int. Conf. on Design and Test in Europe, Proc.*, pages 404–410. IEEE, March 2000.

[Mur96] P.K. Murthy. *Scheduling Techniques for Synchronous and Multidimensional Synchronous Dataflow.* PhD thesis, University of California, Berkeley, CA, 1996.

[NG93] Q. Ning and G.R. Gao. A novel framework of register allocation for software pipelining. In *Symp. on Principles of Programming Languages, Proc.*, pages 29–42. ACM, 1993.

[OH02] H. Oh and S. Ha. Efficient code synthesis from extended dataflow graphs. In *DAC'02, Proc.*, pages 275–280. IEEE, 2002.

[PBB+03] 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 *Int. Conf. on Compilers, Architecture and Synthesis for Embedded Systems, Proc.*, pages 63–72. ACM, 2003.

[SG01] S.K. Shukla and R.K. Gupta. A model checking approach to evaluating system level dynamic power management policies for embedded systems. In *Int. High-Level Design Validation and Test Workshop, Proc.*, pages 53–57. IEEE, 2001.