Latency Minimization for Synchronous Data Flow Graphs

AmirHossein Ghamarian, Sander Stuijk, Twan Basten, Marc Geilen,
Bart Theelen

ES Reports
ISSN 1574-9517

ESR-2007-04
14 June 2007

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

Latency Minimization for Synchronous Data Flow Graphs*

A.H. Ghamarian, S. Stuijk, T. Basten, M.C.W. Geilen and B.D. Theelen
Eindhoven University of Technology, Electronic Systems Group
a.h.ghamarian@tue.nl

Abstract. *Synchronous Data Flow Graphs (SDFGs) are a very useful means for modeling and analyzing streaming applications. Some performance indicators, such as throughput, have been studied before. Although throughput is a very useful performance indicator for concurrent real-time applications, another important metric is latency. Especially for applications such as video conferencing, telephony and games, latency beyond a certain limit cannot be tolerated. This paper proposes an algorithm to determine the minimal achievable latency, providing an execution scheme for executing an SDFG with this latency. In addition, a heuristic is proposed for optimizing latency under a throughput constraint. Experimental results show that latency computations are efficient despite the theoretical complexity of the problem. Substantial latency improvements are obtained, of 24-54% on average for a synthetic benchmark of 900 models, and up to 37% for a benchmark of six real DSP and multimedia models. The heuristic for minimizing latency under a throughput constraint gives optimal latency and throughput results under a constraint of maximal throughput for all DSP and multimedia models, and for over 95% of the synthetic models.*

1 Introduction and Related Work

Synchronous Data Flow Graphs (SDFGs, [9, 10]) have been and are being used widely in modeling data flow applications, both sequential DSP applications [1, 13] and concurrent multimedia applications realized on multiprocessor systems-on-chip [11]. The main goal is to provide predictable performance for those applications. Among the performance indicators, throughput is a prominent one; it has been extensively studied in the literature on SDFGs and related models of computation [3, 4, 6, 7, 14, 17]. Other performance indicators are storage requirements and latency. Buffer minimisation for SDFGs has also been studied [5, 14], but latency has so far only been studied for the subclass *homogeneous* SDFGs [13]. Latency is important in interactive applications such as video conferencing, telephony and games, where latency beyond a certain bound becomes annoying to the users. It is in principle possible to compute latency metrics for an SDFG via a conversion to a homogeneous SDFG, which is always possible [13]. However, this conversion might lead to an exponential increase in the number of nodes in the graph, which makes it prohibitively expensive in any SDFG-based design flow aiming at optimizing and predicting performance metrics such as throughput, buffer sizes, and latency [6, 14]. In this paper, we present latency minimization techniques that work directly on SDFGs. One technique can be used to compute the minimal achievable latency for an SDFG, and it provides an execution scheme that gives the minimal latency. Another, heuristic technique optimizes latency under a throughput constraint. Although it does not always result in the minimal achievable latency, it does give optimal results for all our experiments on six real DSP and multimedia models, and for over 95% of our synthetic models.

The nodes of an SDFG, called *actors*, communicate with *tokens* sent over the edges, called *channels*. Actors typically model application tasks and edges model data or control dependencies. Each time an SDFG actor fires (executes), it consumes a fixed amount of *tokens*, units of control or data, (in a fifo manner) from its input edges and produces a fixed amount of tokens on its output edges. These amounts are called the rates. The production rate of tokens on a channel may differ from the consumption rate on the channel. Therefore, SDFGs are well suited for modeling multi-rate systems. A homogeneous SDFG is an SDFG with all rates equal to one, i.e., it models a single-rate system. If SDFGs are used to analyse the timing behavior of an application, actors are typically annotated with an execution time.

Usually, the dependencies in an SDFG allow some freedom in the execution order of actors. This order determines performance properties like throughput, storage requirements, and latency. An important strength of SDFGs is that they are statically analyzable. This means that the execution order of actors, both for single- and multi-processor implementations, can be fixed at design time via a scheduling scheme, targeting for example minimal buffer sizes or optimal throughput. The class of scheduling schemes providing a static actor execution order is called the class of static order schedulers. An overview of SDFG scheduling techniques can be found in [13].

In this paper, we present a technique to compute the minimal achievable latency between the executions of any two actors in an SDFG. We also present an execution scheme that defines a class of static order schedules that provide minimal latency. Since this scheme may negatively affect throughput, we propose a heuristic to minimize latency under a throughput constraint. We evaluate our schemes in a single-processor context and in a multi-processor context with sufficiently many resources to maximally exploit parallelism, for various buffering schemes. In the multi-processor context, we compare our execution schemes with static order schedules in which all actors fire as soon as they are enabled, called self-timed execution, which is known to provide maximal throughput [13]. In many cases, substantial gains in latency are possible. It further turns out that for

*This work was supported by the Dutch Science Foundation NWO, project 612.064.206, PROMES.

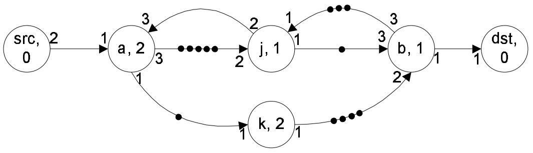

Figure 1. An example SDFG.

all real models and for most synthetic cases minimal latency and maximal throughput can be achieved simultaneously. We also prove that simultaneously optimal throughput and latency is not achievable in all cases.

The next section provides basic definitions for SDFGs. Section 3 defines a notion of latency for SDFGs, generalizing a definition of latency for homogeneous SDFGs [13]. Section 4 introduces an execution scheme that minimizes latency, while Section 5 presents latency-optimal static order scheduling policies for single-processor systems and for a multi-processor context with sufficient resources to exploit maximal parallelism. Section 6 gives our technique for minimizing latency under a throughput constraint. It also disproves the existence of a general scheduling scheme for achieving simultaneous optimal throughput and latency in all cases by providing a counter example. In Section 7, we experimentally evaluate our techniques. Section 8 concludes.

2 Synchronous Data Flow Graphs

Let $I\!\!N_0 = \{0, 1, \ldots\}$ (and $I\!\!N = I\!\!N_0 \backslash \{0\}$) denote the natural numbers. An SDFG G is a pair (A, C) where A is a set of actors, and $C \subseteq A^2 \times I\!\!N^2$ is a set of channels. Each channel $(s, d, p, c) \in C$ denotes that actor s communicates with actor d where p and c are production and consumption rates, respectively. Channels connecting actor s to some other actor d are called output channels of s and input channels of d. An SDFG where all rates are one is called a homogeneous SDFG (HSDFG).

A (channel) state S of G is a mapping $S : C \mapsto I\!\!N_0$ that associates to each channel the number of available tokens on that channel. Each SDFG has an initial state denoted by S_0, providing the number of initially available tokens on each channel.

To enable performance analysis, an SDFG is annotated with timing information. A *timed SDFG* is a triple (A, C, E), with (A, C) an SDFG and $E : A \mapsto I\!\!N_0$ an execution time mapping that associates to each actor $a \in A$, the amount of time $E(a)$ that it needs for firing, the *execution time of* a.

We use the SDFG example depicted in Figure 1 as our running example. It has six actors src, dst, a, b, j, k, denoted by circles containing the actor name and its execution time. Arcs represent channels, and are annotated with production and consumption rates. Tokens are depicted as black dots.

The execution of an SDFG is defined based on actor firings, which may change the state of the SDFG, and hence determine the (reachable) state space of the SDFG. An actor $a \in A$ of SDFG (A, C) is enabled in state S_i if S_i contains at least c tokens for each input channel (s, a, p, c) of a. An actor can fire when it is enabled. Firing a changes state S_i into S_{i+1}, consuming c tokens from each input channel (s, a, p, c) and producing p tokens on each output channel (a, d, p, c).

A timed state of a timed SDFG $G = (A, C, E)$ is a pair (S, τ), with S a channel state and τ the accumulated time. The initial timed state of G is $(S_0, 0)$. A *timed execution* of G is a (finite or infinite) sequence of timed states (S_0, τ_0), $(S_1, \tau_1), \ldots$ where $\tau_{i+1} \geq \tau_i$. Each two consecutive timed states correspond to the firing of an actor a that started its firing in $\tau_{i+1} - E(a)$ and finishes its firing in τ_{i+1}. An execution in which all actors fire as soon as they are enabled is called a *self-timed execution*. Actor firings take time in a timed SDFG which means that they are not atomic. We assume, conservatively, that changes in channel states due to actor firings happen at the end of those firings.

Figure 2 shows a scheduling trace of the running example. The horizontal axis represents the time progress. Each row is dedicated to the firing sequences of an actor. To actors with simultaneous firings more rows are dedicated. For example, actors j and k have two rows each. Each actor firing is represented by a box which starts at the time where the firing starts and lasts as long as the execution time of that actor. As the execution times of actors src and dst are zeros, they are shown by very small boxes. All the executions shown in this article are periodic, and the periodic part which repeats forever is specified between two vertical lines. The latency between actors a and b for the execution of Figure 2 of the running example equals 7, being the total delay between the firings of actors src and dst in any period. It turns out below that the shown execution minimizes the latency between firings of actors src and dst (and hence between firings of a and b).

Only SDFGs satisfying the structural property of *consistency* are of interest. Inconsistent graphs either deadlock or need unbounded channel capacities. Consistency can be verified efficiently [9, 2]. A (timed) SDFG $G = (A, C(, E))$ is consistent if and only if it has a non-trivial repetition vector. A repetition vector for G is a function $q : A \rightarrow I\!\!N_0$ such that for every channel $(s, d, p, c) \in C$, the so-called balance equation $pq(s) = cq(d)$ holds. A repetition vector is non-trivial iff it has no zero entries. The smallest non-trivial repetition vector of a consistent SDFG is referred to as *the* repetition vector. An *iteration* is a set of actor firings with as many firings as the repetition vector entry for each actor. The repetition vector q of the running example equals $(src, a, j, k, b, dst) = (1, 2, 3, 2, 1, 1)$, and the period of the execution shown in Figure 2 consists of precisely one iteration.

3 Latency

This section formally defines a notion of latency for timed SDFGs. Generally speaking, latency is the time delay between the moment that a stimulus occurs and the moment that its effect begins or ends. In timed SDFGs, stimuli are

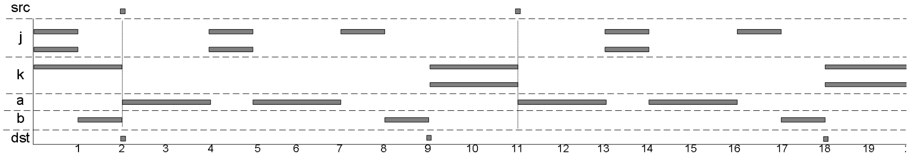

Figure 2. A timed execution of the example SDFG.

actor firings and their effects are the consumptions of produced tokens by some other actors. In the remainder of this article we limit ourselves to consistent, live and strongly connected SDFGs. The latency concepts developed in the article can be extended to non-live and non-strongly connected graphs, but the definitions and reasoning becomes much more tedious. Furthermore, non-live SDFGs are of little interest in the multimedia domain, and many SDFGs in that domain are strongly connected either inherently or due to buffering constraints (See [14]). To define latency, first, we need to define the following.

Definition 1. *[Corresponding Firing] Let $a_1, a_2, \ldots, a_k \in A$ be actors of a timed SDFG (A, C, E) on a path $a_1, a_2, \ldots, a_k$ connecting a_1 to a_k. We say that the j_1-th firing of a_1 corresponds to the j_k-th firing of a_k iff j_2 is the first firing of a_2 which consumes at least one token produced by the j_1-th firing of a_1, j_3 is the first firing of a_3 which uses at least one token produced by the j_2-th firing of a_2, and so on. We denote the firing of a_k corresponding to the j_1-th firing of a_1 by $cf(a_1, j_1, a_k)$.*

Note that in general the time that tokens need to travel from some source actor to some destination actor may differ in different firings of the source actor. In an HSDFG, where all production and consumption rates are one, there is a one-to-one correspondence between actor firings of some source and some destination. Because of differing firing rates, this correspondence does not exist, in general, between actors in an SDFG. In order to arrive at a proper definition of latency for SDFGs, we add an explicit source actor to the source of our latency measurement and a destination actor to the intended destination, each of which fires by construction exactly once in every iteration of the graph. If an SDFG already has meaningful input and output actors with repetition vector entries of one, these actors can function as source and destination and no actors need to be added.

Definition 2. *[Latency Graph] Let $a, b \in A$ be two actors of a timed SDFG (A, C, E) with repetition vector q, and let $src, dst \notin A$ be two new actors. We define the latency graph for actors a and b as $G_{L(a,b)} = (A_L, C_L, E_L)$, where $A_L = A \cup \{src, dst\}$, $C_L = C \cup \{(src, a, q(a), 1), (b, dst, 1, q(b))\}$, and $E_L = E \cup \{(src, 0), (dst, 0)\}$.*

The latency between two actors is defined through the latency of different firings of actors src and dst in the latency graph. Note that src and dst have execution time 0, so that their addition does not influence the timing behavior of the graph. Observe that the example of Figure 1 shows in fact the latency graph for actors a and b of the SDFG obtained when omitting the src and dst actors. In this example, as src does not have any input channel it can fire as often as needed; therefore it puts no restriction on the firings of a. Also, as channels are unbounded, the firing of actor b is not restricted by actor dst. Furthermore, because both actors src and dst have execution times zero, and do not impose any restrictions on the firings of the other actors, any execution of the latency graph is an execution of the original graph too when src and dst are omitted from that execution. The following proposition shows that there is a one-to-one correspondence between src and dst firings (where dst may have some initial firings without corresponding src firing).

Proposition 3. *Let $G_{L(a,b)}$ be some latency graph. There is some $\delta \in I\!N_0$ such that the k-th firing of source actor src for arbitrary $k \in I\!N$, corresponds to the $(k + \delta)$-th firing of dst, i.e., $cf(src, k, dst) = k + \delta$ for all $k \in I\!N$.*

Proof. According to the definition of the repetition vector and of an iteration, we know that if a graph executes a complete iteration i.e., each actor a fires as many as $q(a)$, then the channel state does not change. Therefore, firing a complete iteration, does not change the relation of corresponding firings of any two actors in the graph. Because src and dst each have entry 1 in the repetition vector, the corresponding firing of the k-th firing of src is always firing $k + \delta$ for some fixed δ, if δ is the dst firing corresponding to the first firing of actor src. □

In practice, we are mostly interested in the latency of actors which are considered the input and the output of the system, and these actors often have a repetition vector entry of one already. Furthermore, usually, only executions of complete iterations of graphs are meaningful. Therefore, in case actors have repetition vector entries different from one, we do not look at all firings of those actors. Instead, via the addition of the src and dst actors, the latency is defined on the groups of firings of each actor that contain as many firings of the actors as their repetition vector entries.

In the following, $F^{\sigma}_{a,k}$ represents the finishing time of the k-th firing of an actor $a \in A$ in execution σ. Furthermore, due to resource constraints, such as for example a limited number of processing units, some executions might not be feasible. The set of feasible executions is denoted FE.

Definition 4. *[Latency] Let $a, b \in A$ be two actors of a timed SDFG (A, C, E) with latency graph $G_{L(a,b)}$. The k-th latency of a and b for an execution σ is defined as the time* *delay between the k-th firing of src and its corresponding firing of dst in σ, and it is denoted by $L_k^\sigma(a,b)$:*

$$L_k^\sigma(a,b) = F^\sigma_{dst,cf(src,k,dst)} - F^\sigma_{src,k}.$$

The latency *of actors a and b in execution σ, $L^\sigma(a,b)$, is defined as the maximum k-th latency of a and b for all firings of a:*

$$L^\sigma(a,b) = \max_{k\in\mathbb{N}} L_k^\sigma(a,b).$$

The minimal latency *of actors a and b, $L^{min}(a,b)$, is defined as the minimum over all* feasible *executions in FE:*

$$L^{min}(a,b) = \min_{\sigma\in FE} L^\sigma(a,b).$$

Note that this definition implies that latency is measured from the start time of a firing of actor a (or group of firings), intuitively corresponding to the consumption of some input, to the finishing time of the corresponding (group of) firing(s) of actor b, intuitively corresponding to the production of output directly related to the consumed input. The current definition is consistent with and generalizes the definition of latency given for HSDFGs in [13]. The latency between actors a and b for the execution of Figure 2 of the running example equals 7, being the total delay between the firings of actors src and dst in any period.

4 Minimum Latency Executions

In [6], a technique to compute throughput of SDFGs based on a state-space traversal is presented, showing that this can be done very efficiently in practice, despite the potentially exponential size of the state space. Therefore, in this section, we introduce an execution scheme to determine the minimal possible latency. As an immediate by-product, we obtain a class of static order schedules that achieve minimum latency. We restrict ourselves to strongly-connected graphs. In practice, this is not a restriction, because all SDFGs that can be executed within bounded memory can be turned into strongly connected graphs by modeling channel capacity constraints via backward channels [14].

Definition 5. *[Minimum Latency Execution] Let $G_{L(a,b)}$ be the latency graph of a strongly connected timed SDFG $G = (A, C, E)$ with actors a and b. A feasible execution consisting of the repetition of the following four phases is called a minimum latency execution.*

Phase 1 *Execute actors except src until src is the only enabled actor. (Note that src is always enabled because it does not have any inputs.)*

Phase 2 *Fire src once.*

Phase 3 *Execute, without any unnecessary delays, the minimum set of required actor firings for enabling dst for one firing.*

Phase 4 *Fire dst once.*

Let P_n with $n \in I\!N$ the part of the execution trace which represents the n-th execution of the four phases.

Figure 2 shows a minimum latency execution of the running example. In Phases 1 and 3, execution is self-timed (see Sec. 2). Note that the above execution scheme explicitly schedules the src and dst actors, which is typically possible for DSP and multimedia applications. Also note that an SDFG may exhibit more executions that realize minimum latency than those defined in Definition 5. However, the defined executions are guaranteed to have minimum latency.

Proposition 6. *Let $G = (A, C, E)$ be a timed SDFG with $G_{L(a,b)}$ the latency graph for actors a and b in A. Any minimum latency execution of $G_{L(a,b)}$ has the following properties.*

1. *P_n equals one iteration for all $n > 1$ and the state reached after Phase 1 is the same for all $n \geq 1$.*

2. *The n-th firing of src and its corresponding firing $cf(src, n, dst)$ of dst occur in the same P_n.*

3. *The set of actor firings between any firing of src and its corresponding firing of dst is the smallest possible set among all executions.*

Proof. Part 1: After Phase 1, no actor is enabled, except src. Strong connectedness of G implies that Phase 1 terminates. The repetition vector entry is one for src, and by construction a firing of actor src enables actor a for $q(a)$ firings, when q is the repetition vector. This number of firings of a enables all successors of a for as many firings as their repetition vector entries, and so on. In fact, there is such an execution trace for all live SDFGs [8]. Since there is only one firing of src, no actor can also be fired more often than its repetition vector entry. Some part of these firings happen in Phase 1 and the rest happen in Phases 3 and 4. While going from the end of Phase 1, through Phases 2, 3, 4 and 1 again, the numbers of firings correspond exactly to the repetition vector and, hence, by the balance equations, the state reached must be the same.
Part 2: During the execution of Phase 1 in P_1, dst fires as often as it can without using any tokens from the first firing of src. Hence, the first firing of dst after this phase depends on a token from the first firing of src after this phase, which is the first firing of src ever. src fires in Phase 2 of P_1 and dst in Phase 4 of P_1. In each of the following P_n, based on Part 1 of the proof and the fact that both src and dst have repetition vector entry one, both src and dst fire once in Phases 2 and 4 respectively and because of Proposition 3, these firings are each pairs of corresponding firings.
Part 3: By definition of Phase 1, we know that none of the firings of Phase 3 can fire in Phase 1, since all of them depend on the firing of src. Besides, Phase 3 only fires the minimum set of actor firings needed for enabling dst after the firing of src. Therefore, the set of actor firings in Phase 3 is the minimum set of firings possible between the corresponding src and dst. Hence, for any other execution of the

SDFG, the firings between the corresponding src and dst firings contain at least those firings of Phase 3. □

Proposition 6 shows that the set of firings in between the designated src and dst actors is minimal in any minimum latency execution (prop. 3), that a minimum latency execution is periodic (prop. 1), and that the pairs of corresponding src and dst firings that determine the latency always occur in one period (prop. 2). The precise duration of the firings between src and dst firings depends on the particular execution. The set of allowed executions may be constrained by the available platform; a single-processor platform, for example, does not allow concurrent execution. If Phase 3 firings are executed within platform constraints without unnecessary delays, the following result follows immediately from Proposition 6.

Theorem 7. *[Minimum Latency] Let σ be any minimum latency execution of a latency graph $G_{L(a,b)}$ taken from the set of feasible executions FE. Then, we have*

$$L^{\sigma}(a,b) = L^{min}(a,b).$$

Proof. Part 3 of Proposition 6 states that the set of firings in between a src firing and its corresponding dst firing is the smallest possible. Thus, execution of this necessary set, without any unnecessary delay, leads to an execution with a minimum possible latency. Note that executing a set of actors without any unnecessary delay means that each actor in the set starts its firing as soon as it is enabled and there is a free processor available in the platform on which the graph is executed. In the other words, they are executions in which all actors fire as soon as they have their required data and resources available. □

Observe that Proposition 6 proves that a minimum latency execution has a periodic phase consisting of one iteration of the SDFG. An interesting consequence of this is that code size is limited. In general, executions, such as for example self-timed executions that optimize throughput, might have a periodic phase consisting of multiple iterations [6], which implies a larger code size.

Another interesting observation is that the periodic execution of an SDFG allows for a straightforward computation of the *throughput* of an actor, i.e., the average number of firings of the actor per time unit. In the example execution of Figure 2, the throughput of (output) actor b is $1/9$.

5 Static Order Scheduling Policies

In general, scheduling an application involves assigning actors to processing elements, ordering the execution of each actor on each processing element and finally determining the firing times for each actor such that data and control dependencies are met. In this article, we only look into two types of platform. In both of our platforms, the single-processor and sufficiently-many-processor actor assignments are trivial. Also the last two cases of scheduling in both of these steps are combined by specifying the order of actor firings.

A well-known schedule type in which the firing orders of all actors are determined at the compile time is called the *static order schedule*. The minimum latency execution given in Definition 5 results in a static order schedule as it determines the order of firings of all actors in the execution. In the following, we explain the minimum latency execution for each of the two discussed platforms.

5.1 Single-Processor Scheduling

In the previous section, we have seen that any minimum latency execution leads to a minimum latency between the designated pair of actors. To create a static order schedule for a single processor, it only remains to order the various executions in Phases 1 and 3 of the scheme. If only considering latency, this order can be arbitrary, as long as it satisfies the data and control dependencies specified by the channels in the SDFG. One could decide to try to optimize other constraints such as code size, using for example single appearance scheduling techniques [1, 16]. With respect to throughput, it can be observed that the order in which the individual actors are scheduled in any feasible schedule of a consistent SDFG on a single processor does not impact the average throughput of the application as long as there are no idle periods. Therefore, any minimum latency execution combines minimum latency with the maximal throughput that can be obtained on a single processor.

Figure 3 shows a single-processor static order schedule for our running example that adheres to the minimum latency execution scheme. The latency between actors a and b is 8 and the throughput of b is $1/12$. Both latency and throughput are optimal for a single processor. It is interesting to observe that the minimal achievable latency given some arbitrary amount of processing resources is always between the minimal value as defined in Definition 4 under the assumption that all executions are feasible, which gives the limit imposed by the data and control dependencies in the SDFG, and the minimum latency for a single processor.

5.2 Scheduling with Maximal Parallelism

An interesting case in a multi-processor context, is the case that sufficiently many resources are available to maximally exploit parallelism, or in other words, a context with unlimited processing resources so that any enabled actor can always make progress and all executions are feasible. As mentioned, this allows to determine the minimum achievable latency constrained only by the dependencies in the SDFG. The result can be used as a feasibility check for the application latency in a (multi-processor) design trajectory.

Observe that the crucial point in the 4-phase minimum latency execution scheme is that the actor firings of Phase 1 cannot interfere with the firings in Phase 3. In a single-processor context, this simply means that these two phases have to be executed completely separately. However, in a context with sufficient resources, the two phases can be allowed to execute concurrently, in a self-timed manner (as defined in Section 2), because firings of Phase 1 that are executed concurrently with firings of Phase 3 do not interfere

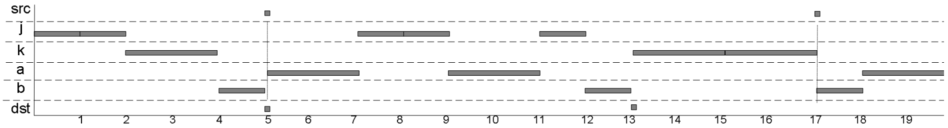

Figure 3. A single-processor minimum latency static order schedule of the example SDFG.

with those Phase 3 firings. Furthermore, self-timed execution minimizes the execution time of the critical path of the actor firings in Phase 3. Since also the firing of dst (Phase 4) can be integrated into this self-timed execution scheme, these observations lead to the following execution scheme.

Definition 8. *[Minimum Latency Execution Scheme with Unlimited Resources]*

Phase 1 *Execute actors of the latency graph except src in a self-timed manner until src is the only enabled actor.*

Phase 2 *Fire src once, and repeat.*

This scheme suggests a concrete multi-processor static order schedule that simply schedules the actor firings in the two phases of this minimum latency execution scheme iteratively in a self-timed manner. Note that the first execution of Phase 1 might be different from the other executions of Phase 1, so that the resulting static order schedule still has a transient part and a periodically repeated part. Figure 4 shows a latency-optimal static order schedule adhering to this scheme. It uses the same conventions as those used for Figure 2. The latency between actors a and b is 7, which is of course the same latency as in the execution of Figure 2 which was already optimal given the dependencies inherent in the SDFG. The advantage of the new execution scheme shows in the improved throughput. The throughput of actor b in the execution of Figure 4 is $1/7$, whereas it is $1/9$ in the execution of Figure 2.

6 Throughput Constraints

A multimedia application is often subject to multiple performance constraints such as latency, throughput and memory usage. So far, we have seen several scheduling policies for obtaining minimum latency. The single-processor policy achieves also the maximum throughput since it fully utilizes the only processing unit. As mentioned, the schedule could be further optimized for code size.

The maximum parallelism policy, as the other policies, disallows overlap between multiple iterations of the SDFG. This has a positive effect on code size, but it potentially influences throughput negatively. By allowing simultaneous firings of the source actor in Phase 1 of Definition 8, multiple iterations of the SDFG execution can be scheduled in parallel, which may lead to a higher throughput [8, Sec. 3.4]. However, this might have a negative effect on latency. In fact, we have the following proposition.

Proposition 9. *[Latency and Throughput Optimization] Given an arbitrary SDFG G; assume sufficiently many resources are available, i.e., the feasibility of executions is only determined by the data dependencies between actors. G does not necessarily have an execution that simultaneously minimizes latency and maximizes throughput.*

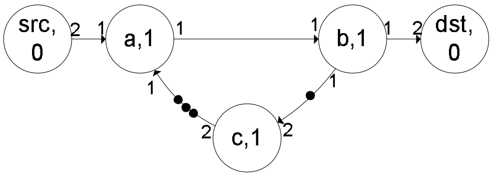

Figure 5. A counter example G for simultaneously optimizing throughput and latency.

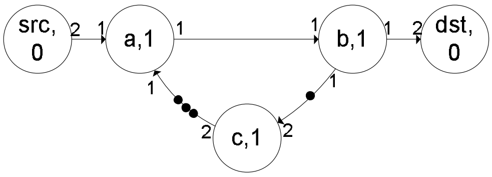

Figure 6. A counter example G for simultaneously optimizing throughput and latency.

Figure 6 shows an example SDFG G for which it is not possible to simultaneously optimize latency and throughput. The minimal latency that can be obtained for G is 2. The minimal latency execution obtained via Definition 8 is shown in Figure 7.

Figure 8 shows the self-timed execution of this example (split in two parts, as explained later), with the exception that actor src fires only when actor a needs tokens for firing. Self-timed execution is known to give maximal achievable throughput [13]. The firings of src in Figure 8 are scheduled in such a way that they do not constrain throughput, so the execution in Figure 8 achieves maximal throughput. For example, the throughput of actor b is $4/3$ firings per time unit. We see that the latency of the execution is 5 (due to the src and dst firings in part (1) of the execution).

The self-timed execution of G can be divided into two parts. Suppose we color the first token on channel c-a and the token on b-c blue and the second and third token on channel c-a red. This coloring implies that firings of actor c always consume and produce tokens of one color. In Figure 8, (1) and (2) correspond to actor firings involving blue and red tokens on channels a-b, b-c, and c-a respectively.

The minimal latency execution of Figure 7 follows the schedule of part (2) in Figure 8, i.e., all tokens are processed

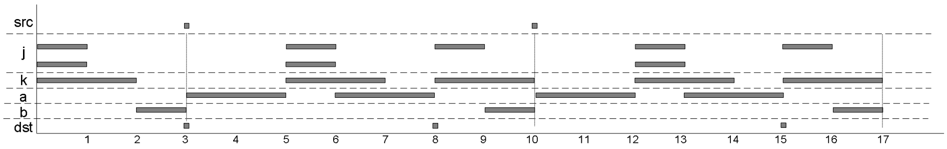

Figure 4. A minimum latency static order schedule using the optimized execution scheme for unlimited resources.

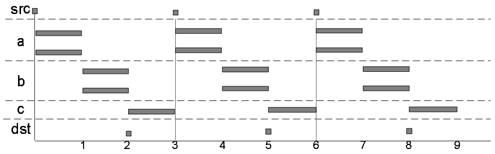

Figure 7. Minimal latency execution of G.

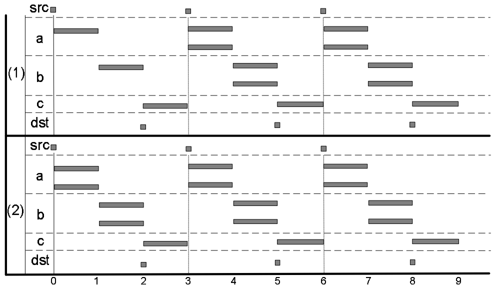

Figure 8. Maximal throughput execution of G.

according to the red scheme. Throughput of b in the execution of Figure 7 is $2/3$ firings per time unit. This is the maximum that can be achieved without executing multiple iterations of G concurrently. (Note that an iteration of G consists of one firing of src, dst, and c, and two firings of a and b.) However, executing multiple iterations concurrently implies that tokens are necessarily processed according to the blue scheme, part (1), of Figure 8 (or an even slower scheme). This implies that increasing throughput necessarily leads to a higher latency, proving Proposition 9.

A consequence of Proposition 9 is that it is interesting to explore throughput and latency trade-off under the maximal parallelism assumption. In the remainder of this section, we propose a heuristic execution scheme that attempts to minimize latency under a given throughput constraint. That is, the algorithm tries to schedule the SDFG in such a way that the throughput constraint is met, while latency is minimized. If the SDFG is inherently too slow to meet the throughput constraint, the algorithm returns a schedule with maximum throughput and a minimized latency.

An important observation is that a throughput constraint can be modeled in an SDFG (See Figure 9). Assume we want to impose a throughput constraint of τ firings per time unit on a designated actor b. This can be achieved by adding a fresh actor tc to the SDFG with a self-loop containing one token to avoid simultaneous firings of tc and with an execution time of τ^{-1}. By adding two channels between b and tc as shown in the figure, tc *on the long run* prohibits b to fire more often than τ times per time unit. We can calculate the number of initial tokens on the channel from tc to b, denoted by k in the figure, such that the graph achieves the maximum throughput that can be obtained by the graph without k-b channel, i.e., such that the cycle through b and tc does not restrict the throughput. In fact, k is the buffer size for the channel connecting b to tc, and the minimum k can be calculated by the method proposed in [14]. Even by choosing the right k the rest of the SDFG might slow down b more than tc, so the realized throughput for b could be lower than τ. The number of tokens in the tc-b channel determines how much short-term deviation in b-s throughput is allowed. This jitter may influence the minimal achievable latency from any source actor to b.

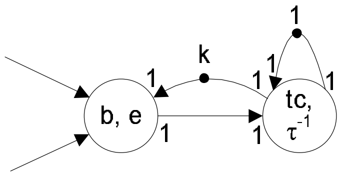

Figure 9. Modeling a throughput constraint.

A throughput constraint can be added to the sink actor of a pair of actors for which latency needs to be minimized. This results in the throughput-constrained latency graph.

Definition 10. *[Throughput-constrained Latency Graph] Let $G_{L(a,b)} = (A_L, C_L, E_L)$ be the latency graph of some SDFG $G = (A, C, E)$ with actors $a, b \in A$ and with repetition vector q. Let τ be a throughput constraint on actor b, and let $tc \notin A$ be a new actor. We define the τ-constrained latency graph for actors a and b as $G_{L(a,b,\tau)} = (A_\tau, C_\tau, E_\tau)$, where $A_\tau = A_L \cup \{tc\}$, $C_\tau = C_L \cup \{c_0 = (b, tc, 1, 1), c_1 = (tc, b, 1, 1), c_2 = (tc, tc, 1, 1)\}$, and $E_\tau = E_L \cup \{(tc, \tau^{-1})\}$. The initial state S_0 for the new channels c_0, c_1, c_2 is defined as follows: $S_0(c_0) = 0$, $S_0(c_1) = q(b)$, and $S_0(c_2) = 1$.*

Given a throughput-constrained latency graph, the goal of minimizing latency under the throughput constraint reduces to minimizing latency while maintaining maximal throughput of the throughput-constrained SDFG. As mentioned, maximal throughput can be achieved via self-timed execution. The algorithm presented below essentially performs a self-timed execution, except that the firings of the designated actor src are delayed. The idea is that latency is minimized by scheduling the firing of src precisely the minimum achievable latency number of time units before the dst firing times in self-timed execution. The algorithm does not change the average number of firings over time of any actor in the graph, although it may delay some firings over time. In other words, the maximal throughput of entirely self-timed execution is maintained, but dependencies in the graph may cause the dst actor to fire at a different moment in time in the schedule produced by the algorithm when compared to the self-timed execution. Consequently, the latency need not be equal to the minimal achievable latency.

```
Algorithm optimizeThroughputLatency ($G_{L(a,b,\tau)}$)
Input: A $\tau$-constrained latency graph $G_{L(a,b,\tau)}$ of a
    strongly connected SDFG $G$.
Output: "A schedule with maximal throughput (under con-
    straint $\tau$) and (close to) minimal latency"
1.  Calculate $L^{min}(a,b)$ from the execution defined in
    Definition 8.
2.  Execute $G_{L(a,b,\tau)}$ in the self-timed manner, and store
    the time of all the firings of actor $dst$.
3.  Execute $G_{L(a,b,\tau)}$ as follows
        - Fire all actors but $src$ if they are enabled.
        - Fire $src$ (which is always enabled) if the time is
          $L^{min}(a, b)$ earlier than the time stored in Line 2
          for the corresponding $dst$ firing.
    return The schedule obtained from the execution of
    Line 3.
```

Theorem 11. *[Maximal Throughput] The schedule returned by algorithm $optimizeThroughputLatency$ achieves maximal throughput under the given constraint τ.*

Proof. Due to consistency and strong connectedness of G, the throughput of all actors in G is in any execution proportionally related through their repetition vector entries as shown in [6]. By construction of the throughput-constrained latency graph, also the throughput of the tc and dst actors is in any execution of $G_{L(a,b,\tau)}$ proportionally related to the other actor throughputs through their repetition vector entries. (The throughput of src is unbounded because it has no input channels.) In the output schedule of $optimizeThroughputLatency$, actor src has by construction exactly the same throughput as actor dst has in the self-timed schedule, which is maximal for dst. Given the above observations and the fact that both src and dst have repetition vector entry one, this implies that src and dst have the same, for dst maximal, throughput in the output schedule. Hence also b has maximal throughput in the output schedule. □

Figure 11 illustrates algorithm *optimizeThroughputLatency* for the running example. The aim is to achieve maximal throughput. In this case, it is not necessary to explicitly model a throughput constraint in the graph. Figure 10 shows the self-timed execution of the latency graph, ignoring actor src (which in principle can fire infinitely often in zero time at time 0 providing unlimited tokens for actor a). This execution is known to provide the maximal throughput for b, which is 1/6. Actor dst fires at times $2 + 6n$ for every $n \in I\!N_0$. The first of these firings does not need a firing of src and can therefore be ignored for latency purposes. Figure 11 shows the output of *optimizeThroughputLatency*. Actor src is scheduled at times $1 + 6n$, i.e., 7 time units (the minimum latency) before every dst firing in the self-timed execution except the first one. The result is a schedule that follows self-timed execution, with the src actor appropriately inserted. It achieves the minimal achievable latency of 7 and the maximal throughput of 1/6. For each src firing, the latency spans the duration till the second subsequent dst firing, i.e., the latency exceeds the length of one period.

Table 1. Results: synthetic, single-processor.

	Min Lat	Arbitr. Order
Strongly Conn. Graphs	4.40	9.65
Min Buff./Throughput	1.36	1.83
Max Throughput	1.31	2.10

Table 2. Results: synthetic, maximal parallelism.

	Latency	Throughput	Execution Time[ms]
Strongly Connected Graphs			
Min latency	1	0.68	5.04
Self-timed	1.43	1	6.29
optThrLat	1.12	1	15.44
Minimum Buffers and Throughput			
Min latency	1	0.90	4.71
Self-timed	1.54	1	6.34
optThrLat	1.10	1	14.14
Maximal Throughput			
Min latency	1	0.78	3.15
Self-timed	1.31	1	3.06
optThrLat	1.01	1	8.73
optThrLat min latency-max throughput: 858/900 (95.3%)			

7 Experimental Results

In this section, we evaluate our scheduling schemes. In case of the single processor scheme, static order schedules with an arbitrary order of the concurrently enabled actors are used as a reference point. In the maximal parallelism scenario, the latency and throughput of the schemes of Definition 8 and of algorithm *optimizeThroughputLatency* are compared with those of the self-timed execution. Since truly arbitrary single processor static order schedules can have a very poor latency, for each SDFG in the experiment, the generated static order schedules were constrained allowing only a single iteration of the SDFG in the periodic part of the schedule, and 100 different static orders were tested, choosing the best result.

We created a benchmark containing six real DSP and multimedia models and three sets of 300 synthetic SDFGs, generated using the SDF^3 tool [15]. The first set is composed of arbitrary strongly connected SDFGs. The second set contains graphs in which the dedicated storage capacity for channels is set to the minimum allowing non-zero throughput (computed via techniques from [14]). The third set contains SDFGs in which the buffer sizes for channels are set to the minimum which is enough to obtain the maximal achievable throughput [14]. All experiments were performed on a P4 PC running at 3.4Ghz.

Table 1 shows results for optimal latency single-processor schedules and the randomly generated static order schedules. The latency entries are averaged over the entire set of models and normalized wrt minimal achievable

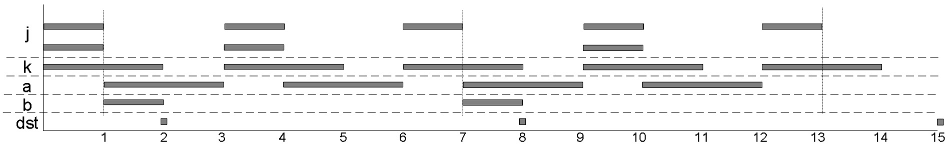

Figure 10. Self-timed execution.

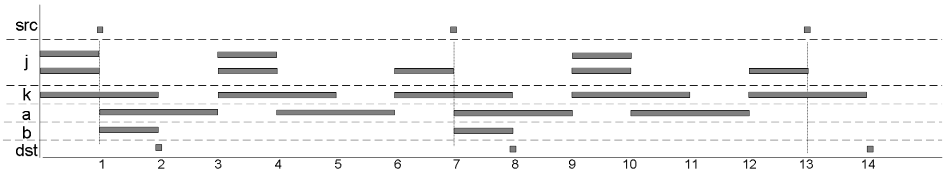

Figure 11. The execution result of *OptimizeThroughputLatency*.

Table 3. Results: DSP and multimedia benchmark.

	Lat (avg/worst)	Thr (avg/worst)	Exec Time[ms]
Single-processor results			
Min latency	$1/1$	$1/1$	3316
Random-order	$1/1.03$	$1/1$	3963
Minimum Buffers and Throughput			
Min latency	$1/1$	$0.67/0.34$	7339
Self-timed	$1.11/1.36$	$1/1$	38262
optThrLat	$1/1$	$1/1$	80410
Maximal Throughput			
Min latency	$1/1$	$0.56/0.32$	7050
Self-timed	$1.11/1.59$	$1/1$	4840
optThrLat	$1/1$	$1/1$	10339
optThrLat min latency-max throughput: 6/6 (100%)			

latency (without the single-processor constraint). Minimum latency execution improves latency between 26% and 54%. Recall that throughput is the same for both techniques.

Table 2 shows, for the synthetic graphs and the maximal parallelism scheme, the latency, throughput and execution time results of minimum latency execution (Definition 8), self-timed execution and latency optimized maximal throughput execution (Alg. *optimizeThroughputLatency*). All entries show the average numbers taken over all 300 graphs of each set. The entries for latency are normalized wrt the results of the minimum latency schedule and the throughput entries are normalized wrt the throughput achieved in the self-timed execution (i.e., the maximal achievable throughput). The self-timed schedule has a 31-54% higher latency than the minimum latency execution. In other words, minimum latency execution gives a latency reduction of 24-35% compared to self-timed execution. The price to be paid is a decrease in throughput of 10-32%. The latency optimized maximal throughput execution reduces the latency with 22-29% wrt self-timed execution, while guaranteeing maximal throughput. The achieved latency is close to the minimally achievable latency (within 10% on average). In over 95% of the graphs the result combines minimal achievable latency with maximal achievable throughput. The average execution time for a single SDFG for any of the scheduling algorithms is a few milliseconds.

We also experimented with SDFG models of actual DSP and multimedia applications. DSP domain applications are a modem and a sample-rate converter from [1], a channel equalizer, and a satellite receiver [12]. For the multimedia domain, we used an MP3 and an H.263 decoder from [14]. Table 3 shows the results, giving both average and worst-case values for latency and throughput.

The single-processor experiments show only a small latency improvement in one case. Due to the limited parallelism in the graphs, 100 randomly generated static order schedules was in five out of six cases sufficient to achieve optimal results.

Under the maximum parallelism scheme, we considered the application models both with minimal buffers for non-zero throughput and minimal buffers for maximal throughput. The average latency improvement of minimum latency execution wrt self-timed execution is 10%, at a throughput loss of 33-44% on average. The satellite receiver, the modem, and the H.263 decoder do not show any improvement. The channel equalizer (26%, minimal buffers) and the MP3 decoder (37%, maximal throughput) show the largest latency improvements (but also the largest throughput loss). However, applying algorithm *optimizeThroughputLatency* to achieve optimal latency for maximal throughput, achieves maximal throughput and minimal latency simultaneously in all cases. Execution times confirm the feasibility of the proposed techniques.

To test the hypothesis expressed in the introduction that latency optimization via a conversion to homogeneous SDFGs is often infeasible, we applied our techniques also to the HSDFG equivalents of our DSP and multimedia models. In two cases (satellite receiver, H.263 decoder), self-timed execution of Phase 2 of minimum latency execution (Definition 5), which in essence for HSDFGs is a critical path analysis taking into account parallel and pipelined execution that any potential HSDFG-based latency optimization technique has to perform, takes several hours. This indeed renders HSDFG-based techniques prohibitively expensive in these cases.

8 Conclusions and Future Work

We have presented a technique to compute the minimum latency that can be achieved between firings of a designated pair of actors of some SDFG. The technique is based on an execution scheme that guarantees this minimum latency. We presented static-order schedules for single-processor platforms, and for a multi-processor context with sufficient resources to maximally exploit the available parallelism in an SDFG. The latter can be used as a feasibility check for application latency in any multi-processor design trajectory. The experimental evaluation shows that the latency computations and the underlying execution schemes are efficient. Compared to traditional scheduling techniques and execution schemes, substantial reductions in latency can be obtained, sometimes at the price of other performance metrics such as throughput. We showed that it is not always possible to simultaneously optimize latency and throughput. Therefore, we also presented a heuristic for optimizing latency under a throughput constraint. The heuristic gives optimal results for both latency and throughput simultaneously for all our real DSP and multimedia models, and for over 95% of our synthetic models. Future work includes the development of scheduling schemes for concrete multiprocessor platforms without exploiting maximal parallelism, either because insufficient resources are available or because inter-processor communication is expensive. We also plan to investigate the trade-offs between latency, throughput, code size and storage requirements in more detail.

References

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

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

[3] A. Dasdan. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *ACM Transactions on Design Automation of Electronic Systems*, 9(4):385–418, October 2004.

[4] A. Dasdan and R.K. Gupta. Faster maximum and minimum mean cycle algorithms for system-performance analysis. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(10):889–899, 1998.

[5] M.C.W. Geilen, T. Basten, and S. Stuijk. Minimising buffer requirements of synchronous dataflow graphs with model-checking. In *42nd Design Automation Conference, DAC 05, Proceedings*, pages 819–824. ACM, 2005.

[6] A.H. Ghamarian, M.C.W. Geilen, S. Stuijk, T. Basten, A.J.M. Moonen, M.J.G. Bekooij, B.D. Theelen, and M.R. Mousavi. Throughput analysis of synchronous data flow graphs. In *6th International Conference on Application of Concurrency to System Design, ACSD 06, Proceedings*, pages 25–36. IEEE, 2006.

[7] R.M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics*, 23(3):309–311, 1978.

[8] E.A. Lee. *A Coupled Hardware and Software Architecture for Programmable Digital Signal Processors*. PhD thesis, University of California, Berkeley, June 1986.

[9] 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.

[10] E.A. Lee and D.G. Messerschmitt. Synchronous dataflow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[11] 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 *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES 03, Proceedings*, pages 63–72. ACM, 2003.

[12] S. Ritz, M. Willems, and H. Meyr. Scheduling for optimum data memory compaction in block diagram oriented software synthesis. In *International Conference on Acoustics, Speech, and Signal Processing, Proceedings*, pages 2651–2654. IEEE, 1995.

[13] S. Sriram and S.S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc, New York, NY, USA, 2000.

[14] S. Stuijk, M.C.W. Geilen, and T. Basten. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *43th Design Automation Conference, DAC 06, Proceedings*, pages 899–904. ACM, 2006.

[15] S. Stuijk, M.C.W. Geilen, and T. Basten. SDF^3: SDF For Free. In *6th International Conference on Application of Concurrency to System Design, ACSD 06, Proceedings*, pages 276–278. IEEE, 2006.

[16] W. Sung, J. Kim, and S. Ha. Memory efficient software synthesis from dataflow graphs. In *International symposium on System Synthesis, ISSS'98, Proceedings*, pages 137–144. IEEE, 1998.

[17] N.E. Young, R.E. Tarjan, and J.B. Orlin. Faster parametric shortest path and minimum-balance algorithms. *Networks*, 21(2):205–221, 1991.