Performance Analysis of Complex Systems by Integration of Dataflow Graphs and Compositional Performance Analysis

Simon Schliecker, Steffen Stein, Rolf Ernst
Institute of Computer and Communication Network Engineering
Technical University of Braunschweig, Germany
[schliecker | stein | ernst] @ida.ing.tu-bs.de

Abstract

In this paper we integrate two established approaches to formal multiprocessor performance analysis, namely Synchronous Dataflow Graphs and Compositional Performance Analysis. Both make different trade-offs between precision and applicability. We show how the strengths of both can be combined to achieve a very precise and adaptive model. We couple these models of completely different paradigms by relying on load descriptions of event streams. The results show a superior performance analysis quality.

1 Introduction

Formal approaches to system performance analysis are gaining industrial importance with increasing system complexity. Research in this area has produced a variety of models of computation to describe and reason about the system's behavior. Each of these established models represents a focus on different system properties or is adapted to specific multiprocessor design styles. Some models may be very well suited to give performance information for a specific setup, and be inexact or inapplicable for another.

The timing of real-time systems remains a large application area for the use of such formal methods. Key metrics are latency and throughput of the system. In this paper, we focus on two analysis methods based on Synchronous Dataflow (SDF) graphs and event model driven compositional performance analysis.

SDF graphs are well suited to describe signal processing applications such as filters, data-driven hardware components, and also software functions. The data-dependencies and parallelism can be well expressed with SDF, but some dynamic paradigms such as priority driven scheduling can not directly be modeled. Determining optimal schedules and computing the average achievable throughput of the described system has been widely discussed in literature e.g. [10, 4, 3]. Less attention has been given to the worst-case traffic and latency of events in such a graph, as required for our mixed setup.

In the case of more dynamic multiprocessor behavior, compositional methods based on local schedulability analysis yield good results. System properties can be derived by composing local analysis instances using event streams describing key integration issues [5, 12]. This approach allows to integrate a wide range of local details but can not evade overestimations between component boundaries.

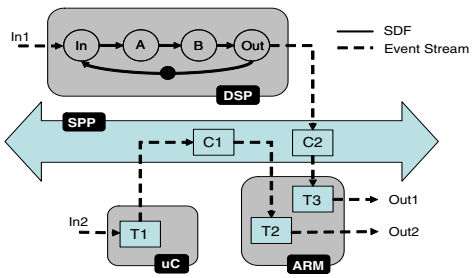

Figure 1. Integrated Setup Overview

Modern embedded systems integrate applications from diverse domains. Take for example the system depicted in figure 1. Here, a DSP, a microcontroller and an ARM processor core are interconnected by a bus. The bus as well as the ARM core are assumed to be arbitrated in a static priority preemptive fashion. A filtering application implemented on the DSP runs a fixed order schedule. The DSP application is best modeled using SDF graphs, whereas schedulability analysis is required for the bus timing as well as for the ARM core.

Integrating both analysis approaches in a single framework as presented in this paper will increase compositional analysis accuracy and extend the applicability of SDF analysis into the domain of systems that contain priority driven components. It allows the designer to choose the most suitable model.

This paper presents the involved models for performance analysis (Secs. 2.1, 2.2), our integrated setup (Sec. 3), and algorithms to derive the required data from an SDF graph (Sec. 4). Experiments are shown in Sec. 5, and we conclude in Sec. 6.

2 Related Work

2.1 Compositional System Level Performance Analysis

The compositional performance analysis methodology proposed in [5] (available as the tool SymTA/S), solves the global system-level performance verification problem by decompos-

978-3-9810801-2-4/DATE07 © 2007 EDAA

ing the system into independently investigated components.

Each resource (as seen in in Fig. 1) is a component and the possible I/O timing between the tasks (*event streams*) is captured with event models. *Input event models* capture event patterns leading to task activations. These are used to perform a local scheduling analysis of a resource to derive the local response times as well as *output event models*. These output event models are propagated to subsequent resources where they are used, in turn, as input event models. In setups with cyclic dependencies the assumed event streams become increasingly more generic. This procedure either converges (and provides a conservative estimation of system properties such as jitter and latencies which can be checked against given constraints), or the system's schedulability can not be guaranteed.

Fundamental properties of the event models used in [12] and [5] are the maximum and minimum number of events that may occur during a time interval of given size. Correspondingly, the minimum $\delta^-(i)$ and maximum distance $\delta^+(i)$ between successive events can be used. To receive a more compact description, these models are expressed in [9] with a set of key parameters: the period $\mathcal{P}$, the signal jitter $\mathcal{J}$, and the minimum distance between any two events d^{min}. The distance for $i \geq 2$ events in a bursty event stream ES can then be expressed as follows:

$$\delta^+(n) = (i-1)\mathcal{P} + \mathcal{J} \quad (1)$$

$$\delta^-(n) = \max((i-1)d^{min}, (i-1)\mathcal{P} - \mathcal{J}) \quad (2)$$

Event models only capture key integration aspects such as send/receive message jitters, etc., and ignore details of the concrete computation or communication resources. They therefore represent a suitable abstraction for the integration of other analysis methodologies or application models.

2.2 Synchronous Dataflow Graphs

A Synchronous Dataflow graph is a directed graph. The nodes of the graph model tasks and are referred to as *actors*. The edges model communication channels or other dependencies between actors. *Tokens* on the edges represent data samples that are consumed or produced by the execution (*firing*) of an actor. An actor may consume or produce more than one token upon each firing. The actual number of consumed or produced token is called *consumption* or *production rate* respectively. If the consumption and production rates of all actors equal one, the graph is referred to as a Homogeneous SDF (HSDF) graph.

The classical SDF model [7] is untimed. However, fortiming analysis purposes, SDF Graphs have been annotated with execution times for each actor [10, 4]. Many problems related to SDF Graphs have been thoroughly discussed in literature. This includes computation of the throughput of an SDF Graph which involves calculating the Maximum Cycle Mean λ of the graph [3]. In literature, given SDF graphs are often converted to equivalent homogeneous SDF (HSDF) [10] graphs to simplify reasoning about them. The conversion algorithm given in [10] also yields a repetition vector $\vec{r}$, indicating how many firings of each node will return the graph into the initial state. Also, $\vec{r}$ indicates for each node, how many copies of it will be present in the equivalent HSDF graph. As the HSDF graph can be considerably more complex, recent work [4] directly derives throughput from the SDF representation.

SDF graphs can be used to express the (partial) order of the application's data dependencies. Likewise they can accommodate additional constraints imposed by shared resources and scheduling. This is done to reason about and derive an optimal static order schedule e.g. in [10]. A much larger application area is described in [8], where the self-timed execution of an SDF model of the actual system is used to also model the delay of computation, communication, and arbitration of suitable hardware components. Under self-timed execution each actor fires as soon as all input tokens are available. Reasoning about this model delivers conservative estimates of the actual hardware behavior. In this paper we assume that an SDF model of the application exists for which the previous is true.

SDF graphs under self-timed execution have been shown to behave monotonically, which means that the earlier arrival time of a token can not lead to the later activation of any actor. Furthermore, it is shown that *consistent* graphs that do not *deadlock*, will after a transient phase settle into a *periodic regime* [10]. During the periodic regime, each actor n will fire $N * r[n]$ times until the graph returns into its initial state [1]. N can be derived from the number of tokens on the critical cycles in the graph and is well bounded in most practical cases [10].

The state of the self-timed execution of an SDF graph is defined by the current token placement and execution progress of the involved actors. If we explicitly derive an initial state (as in Secs. 4.3, 4.4), we reduce the state space by representing it by a token placement alone as in [4]. Any state can be reduced to such a simplified state, by either moving all unfinished tokens to the input or to the outputs of the processing actor.

2.3 Mixing Models

Previous work has investigated the composition of performance analysis methodologies from different domains. In [2], a mixed simulation setup is presented that embeds a dataflow graph into a discrete event setup and vice-versa. Many other domains have been coupled in the Ptolemy project. In [6] a simulated component is integrated into a formal real-time performance analysis method. This allows to regard features which are difficult to capture with formal models on the one hand, and speed up analysis time by abstracting detailed behavior on the other. Because they rely on simulation, both of the above approaches do however not derive worst-case guarantees for the integrated components.

3 Embedding SDF into Compositional Performance Analysis

With the tracking of specific event timing, SDF graphs are a suitable model for tightly integrated systems with feedback loops and many cyclic dependencies. On the other hand, the compositional performance analysis can address the complexity of larger systems with abstract event streams and decoupled

local analyses. Exclusive use of either model (if possible) either results in a large computational complexity or the potential loss of precision.

We propose a technique to embed components modeled with an SDF graph under self-timed execution into the compositional analysis. As input load to the SDF model, we allow arbitrary event models that express deterministic and conservative traffic bounds (as in Sec. 2.1). For a successful integration as in Fig. 2, the following information needs to be exchanged:

1: The current *input event models* to the SDF component are required to analyse its timing behavior. They are provided by the compositional performance analysis (of Sec. 2.1).

2: The *rate relationships* of the nodes in the SDF graph demand a relationship between the event streams at the inputs. Furthermore, it must be assessed whether the input load can be processed by the subsystem (see Sec. 4.1).

3: To further reason about the behavior of the complete system, the load imposed on subsequent components is derived (and represented by *output event models*, see Secs. 4.2, 4.3).

4: Finally, reasoning about system properties such as end-to-end delay requires information about the *latency* of events processed by the SDF component (see Sec. 4.4).

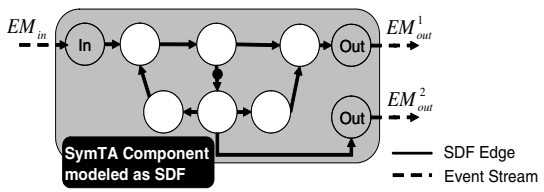

Figure 2. Interfacing of SDF Model

We will use the following notation: Each actor n of the SDF graph G is annotated with an execution time interval $[E_{min}, E_{max}]$. The SDF component has *input ports* $\mathbf{P}_{in}$ and *output ports* $\mathbf{P}_{out}$. In the compositional domain, these are connected to *input event streams* $ES_p, p \in \mathbf{P}_{in}$ and *output event streams* $ES_p, p \in \mathbf{P}_{out}$. Events arriving on an input event stream ES_p are translated to tokens placed on an *edge* e_p of the SDF graph. Equivalently, tokens produced on an output edge translate to events of the corresponding output event stream. We talk about *tokens* and their arrival times in the SDF graph and *events* in the compositional domain. The arrival time of the i-th token on an SDF edge e_p is denoted with t^i_p. The minimum (maximum) distance between i sucessive events of an event stram ES_p is denoted with $\delta^-_p(i)$ ($\delta^+(i)$). An event stream ES_p does on average not produce more than $\mathcal{P}_p$ events (which is its period).

Also, some assumptions are required for the further reasoning: We assume the SDF graph to be strongly connected, which means that a directed sequence of edges exists from any actor to any other. Any setup with bounded buffers can be modeled using a strongly connected SDF graph [10], hence this restriction has no major impact on the usability of the model. Further, we assume all edges and actors retain a first-in-first-out order of the processed tokens.

4 Deriving Performance Metrics from SDF

4.1 Input Period Consistency

The input event streams connected to an SDF graph are constrained in two ways: It must be ensured that incoming taffic can be processed by the component and the long term average of the inputs must be conform with the rate relationships demanded by the SDF graph. Evaluating these properties before continuing with an analysis can significantly reduce runtime, since computation of the latency and output event models can possibly be avoided.

During the periodic regime of the self-timed execution of an SDF graph, each actor $n \in G$ fires $N * r[n]$ times during a macro period of time $N * \lambda$ (see Sec. 2.2). Assuming an input actor consumes c tokens on each activation an actor can on average consume $\frac{c*r[n]}{\lambda}$ from an input edge e_{in}. An input event stream ES_{in} must on average not produce more tokens than can be consumed in order not to overload the graph, which yields the following constraint on the input period:

$$\mathcal{P}_{in} \geq \frac{\lambda}{c_{in} * r[n_{in}]} \tag{3}$$

where λ is computed assuming maximum execution times for each actor.

If the SDF graph has multiple inputs, the period of each of the event streams connected to them has to comply with equation 3. Since the graph is strongly connected, each connected event stream will restrict the complete system in means of bandwidth achievable for other inputs. In order to not overload the graph, the input periods must pairwise comply with the following restriction:

$$\frac{\mathcal{P}_{in,1}}{c_{in,1} * r[n_{in,1}]} = \frac{\mathcal{P}_{in,2}}{c_{in,2} * r[n_{in,2}]} \tag{4}$$

4.2 Output Period

As discussed above, the event streams connected to the SDF graph will constrain its achievable throughput. Similar to computing the maximum period of the output actor dependant of the MCM of the graph, the output period of a given actor n_{out} can be computed from $\mathcal{P}_{in}$, $\vec{r}$, and the number of tokens consumed by the input (c) and produced by the output node (p) on each activation.

$$\mathcal{P}_{out} = \frac{r[n_{in}] * c}{r[n_{out}] * p} * \mathcal{P}_{in} \tag{5}$$

As all $\mathcal{P}_{in}$ comply with equation 4 the above equation yields the same result for each input event stream.

4.3 Output Jitter and Minimum Distance

To construct an output jitter from an SDF component connected to a surrounding SymTA/S model, we analyse corner cases of the load imposed by the SDF component.

First, we analyse a scenario that produces the maximum

load on a given output edge p during self-timed execution of the SDF graph. The observed token production times directly translate to the minimum and maximum inter-event times $\delta_p^-(i)$ and $\delta_p^+(i)$ of an input event stream. To receive an event model conform with the description in Sec. 1 and [9], we derive the jitter from the lower and upper amount of observed tokens.

In this paper, we focus on the maximum load a SDF component can impose on its environment. For completeness, we also present a simple method to derive the minimum load imposed.

Maximum Output Load To construct a starting configuration yielding the maximum output load for a given actor n, we evaluate a set of equations describing the activation timepoints of all actors in an HSDF Graph. Let π_n be the set of predecessors of n, $d(e(j,n))$ denote the initial number of tokens on the edge from j to n, and $a_n(k)$ denote the kth activation of n. Then

$$a_n(k) = \max_{j \in \pi_n} \{a_j(k - d(e(j,n)) + E(j)\} \tag{6}$$

with

$$a_n(k) = -E(n)\ \forall k < 0 \vee E(j) \in [E_{min}, E_{max}] \tag{7}$$

captures the initial state of the graph and all future activations ($k \geq 0$) of an actor n (corresponding to the evolution equations of [1]). Eq. 6 states that an actor becomes activated as soon as there are tokens on all its input edges and models the dependencies between successive actor activations. The first part of Eq. 7 provides that the initial tokens have been produced at time zero. This setup reduces the set of possible initial states to those with no ongoing computation. Any diverging initial state can conservatively be mapped to this set as in Sec. 2.2.

The above equations describe the evolution of the states of the SDF graph during self-timed execution from a given start configuration. Maximizing the load induced by a given output actor n implies maximizing the density of produced tokens over time. Due to monotonicity, a maximum density will be reached for minimal execution times of all actors. Assuming constant execution time for all actors, production of a token on an output edge can be related to activation of the output node by a static offset (its minimum execution time). Thus, the problem of finding a maximum density for produced tokens on an output edge is equivalent to maximizing the activation density of the output actor. Then, the problem of finding the maximum load at an output can be formulated as

$$minimize\ a_n(k) - a_n(0)\ \forall k \geq 0 \tag{8}$$

This way we reformulated the problem of maximizing the output load of a given node to minimizing equation 6 for all $k \geq 0$ and a given n by altering the initial token placement. It is hard to find the reachable token placement that minimizes equation 8 for all k. Thus, we introduce a method to derive a token placement that will overestimate all reachable placements in minimizing equation 8.

It is easy to see that any actor n will instantly fire most often, if all input edges initially bear the maximum amount of tokens possible. For more tokens to be produced on the incoming edges of n as soon as possible, the same argumentation applies to the predecessors of n and their input edges. Due to the cyclic nature of the system of equations [1], this chain of argumentation can be continued until all firings of nodes are dependant on the first firings of actor n. Thus minimizing the timepoints of these will minimize all activation timepoints of n. A solution that will yield minimum activation times for n is to put the maximum number of tokens on the edges with increasing distance from n.

The maximum number of tokens placeable on any edge is well bounded in a strongly connected HSDF graph, as the number of tokens in a simple cycle must remain constant [10].

This constraint limits the number of initial token placements to a superset of the actually reachable token placements. Further constraints taking into account i.e. actor execution times and properties of the connected event streams will tighten this approximation and improve the results of the algorithms to calculate the upper envelope described below.

It can be shown that the method outlined above yields a unique token placement for all HSDF graphs [11]. Algorithm 1 fires all nodes except the output as often as possible, resulting in the token placement yielding the highest output load during self-timed execution.

Algorithm 1 Construct Token Placement

```
INPUT: HSDF graph G, Output node n ∈ G
OUTPUT: Token placement will produce maximum output load
  oneFired = true
  while oneFired do
    oneFired = false
    for all nodes v ∈ G\{n} do
      if v can fire then
        fire v
        oneFired = true
```

To find the actual activation pattern of the output node given the above initial token placement, we assume all input event streams to produce tokens at a rate corresponding to their maximal load $\delta^-(i)$ and simulate the resulting graph assuming minimal execution times for all actors. The simulation can be stopped once the graph has entered the periodic regime [10].

From the observed timepoints t_{out}^k at which the k-th token was produced on the output edge, we construct a function $\delta(i) = t_{out}^i - t_{out}^1$, describing the distances between the observed tokens. $\delta^-(2)$ equals the d^{min} parameter used in the event model of [9]. Since d^{min} and $\mathcal{P}$ are known, the jitter can be easily discovered from the distances $\delta^-(i)$.

Minimum Output Load The minimal load an SDF component can impose on the connected SymTA/S model can be derived from the maximum length of time during which no token has to occur at all. This time span is conservatiely approximated by the difference between maximum and minimum latency (as introduced in the Sec. 4.4). From this, a jitter as necessary for the event model of [9] can be deduced from $L_{max} - L_{min} = \mathcal{P} + \mathcal{J}$.

This approach is trivial. For improved considerations, it can be replaced by a similiar procedure as undertaken for the maximum load.

4.4 Path Latency

End-to-end latency is a fundamental metric to quantify system performance. As we model a subsystem with SDF, we require the local latency for further system wide considerations. The *path latency* as the time between the arrival of an event at an input and the *first causally dependant* reaction at the output. A token x is causally dependant on all tokens that are consumed with the actor activation that leads to the production of x.

The maximum latency is difficult to predict from an SDF representation directly. We therefore assume to have an HSDF representation which duplicates the actors according to the repetitions vector (see Sec. 2.2). We assume for the remainder of this paper that the input and output actors of the path, whose latency we are interested in, are not duplicated. This is the case for many applications. In other cases, the worst case latency is given by the maximum over all latencies from any corresponding input actor to any corresponding output actor.

We are now interested in the latency of an arbitrary token, that arrives at an input edge e_{in} at time t_{in}^0. The latency to an output edge e_{out} can be derived by simulation of the self-timed schedule [10]. The first causally dependent reaction at the output can easily be tracked with a colored token at the input that progressively causes produced tokens to be colored as well. It will be observed at the output after it has propagated over a path from e_{in} to e_{out}. Let $\rho(e_{in}, e_{out})$ be the minimum number of tokens on any such path at time t_{in}^0, then the $1+\rho(e_{in}, e_{out})$-th firing of the output actor after the arrival of the token at the input must be causally dependant. Thus, the latency for the token arriving at the input is given by

$$L \quad = \quad t_{out}^{1+\rho(e_{in},e_{out})} - t_{in}^0 \tag{9}$$

To derive the maximum latency, we assume the graph in a worst case state at t_{in}^0 and let it behave as slow as possible thereafter. As the arrival time of the token t_{in}^0 is fixed, we maximize its latency by maximizing $t_{out}^{1+\rho(e_{in},e_{out})}$.

From Eq.6 it can immediately be seen that for any actor j decreasing actor execution times (E_j) or activation times (a_j) can not lead to a later activation (and therefore finishing) of another (a_n). This is true throughout the graph. Therefore, $t_{out}^{1+\rho(e_{in},e_{out})}$ is maximal under the following conditions: (i) throughout the graph every actor always executes with its worst case execution time (see also [4]), and (ii) events at the graph's inputs arrive as late as possible.

Also, we again do not need to precisely model the state of the HSDF graph at t_{in}^0. We assume the initial state to be defined only by the token placement. Any differing state can be expressed by assuming that tokens currently processed by an actor are placed on the respective incoming edge. This overestimates the resulting latency, but greatly reduces state space.

Assume for now that the only external input event stream is ES_{in}. At the input edge e_{in}, the latest possible arrival of tokens previous to t_{in}^0 are given by the minimum distances inherent to the event stream: $t_{in}^{-1} = t_{in}^0 - d_{in}^-(2)$, $t_{in}^{-2} = t_{in}^0 - d_{in}^-(3)$, a.s.o. Subsequent tokens arriving after t_{in}^0 do not effect the latency due to the FIFO ordering throughout the model.

To produce the worst case state at t_{in}^0, we do however not need to begin the simulation at a time $t_{sim} = t_{in}^{-\infty}$ in the past. For any event stream which can be modeled as in Sec. 2.1, we can assume the additional distance to previous events to be constant after a transient phase. This is e.g. the case for the burst event model of Eq. 1 when the $\mathcal{P}$ term dominates the d^{min} term [9]. In other models this may not be so easy to determine, but it is always conservative to assume constant distances to previous events starting with an arbitrary event arrival t_{in}^{-S}. As the distance increment is non-decreasing, this means that any prior events will be assumed later than is actually possible (and thus lead to a higher latency). Thus, after a number of events S in the past the additional minimum distance to any prior events can be assumed to be constant.

The effect is that the graph that produces the maximum latency at time t_{in}^0 behaved purely periodic at some time in the past. The self-timed execution of an HSDF graph with constant execution times is in the same state after N firings of every actor (once it has entered its periodic regime) [10] . The same is true for our setup before t_{in}^{-S}, as all graph elements have a constant behavior (at their bounds). Thus, the graph will be in the same state at t_{in}^{-S} as it was at $t_{in}^{-(S+N)}$, making it unnecessary to investigate any prior states. Thus, we begin the simulation at $t_{sim} = t_{in}^{-S}$.

The graph can be in any of the N states with the beginning of the simulation. In general, each can lead to different $t_{out}^{1+\rho(e_{in},e_{out})}$ and must therefore be tested with a different simulation run. Alternatively, an approach similar to that in Sec. 4.3 can be used to derive a conservative initial placement.

So far we have neglected other input event streams $ES_p \in \{\mathbf{P}_{in} \setminus ES_{in}\}$ to the graph. On average these inputs will not constrain the graph's execution (see Sec. 4.1), but they may introduce transient *underflow* situations that can increase the searched latency. We create the most constraining scenario by assuming events to arrive as late as possible after the beginning of the simulation: $t_p^0 = t_{sim} + d_p^+(2)$, $t_p^1 = t_{sim} + d_p^+(3)$, a.s.o. for all inputs. This setup puts transient underflow situations at the beginning of the simulation.

Thus, we begin a simulation at a timepoint t_{sim} sufficiently in the past either in each of N graph states or a conservative approximation. We then let external events arrive as late as possible, including the input token at t_{in}^0, and it will experience the worst-case latency, which is determined by the first causally dependant reaction observed at the output.

The minimum latency can be determined in a similar fashion. However, an easy and appropriate simplification is that the first causally dependant effect of an arriving event can not observed before it has propagated at least through the shortest path, where the shortest path is the path with the smallest sum of execution times.

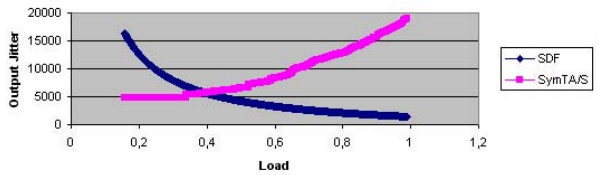

Figure 3. Output Jitter over Resource Load

5 Experiments

The experiments emphasize two aspects of the work presented in this paper. They show that there exists a domain in which SDF analysis of the same system yields better results than traditional performance analysis using SymTA/S. Secondly, the feasibility of the proposed approach is demonstrated by presenting results of a coupled analysis.

For the experiments, we return to the multiprocessor setup shown in Fig.e 1 and assign the actors, computation and communication tasks with execution times as shown in Table 1. Actor execution on the DSP is data driven and priorities on the bus are assumed to be C1 $>$ C2 and on the ARM T3 $>$ T2.

(a) Core Execution Times

Task	CET
T1	[10,20]
T2	[22,500]
T3	[43,132]

(b) Core Comm. Times

Channel	CCT
C2	[76,92]
C1	[92,122]

(c) Execution Times of SDF Actors

Actor	CET
In	[10,200]
A	[50,100]
B	[600,1000]
Out	[60,100]

Table 1. Experimental Setup

To compare the SDF-based analysis with the SymTA/S approach, we use the calculated jitter as a benchmark value. As both approaches yield conservative results, a smaller jitter means tighter analysis of the modeled system. As an experimental setup, we modeled the task-chain on the DSP both in SDF as well as SymTA/S. In SymTA/S, the task chain was modeled as a task-chain mapped on a round-robin scheduled resource.

In an experimental run, we increased the load on the DSP resource by decreasing the period of an input signal and compared the output jitters as calculated using the proposed SDF analysis technique as well as using the SymTA/S toolsuite. The results are summarized in Fig. 3. It can be seen, that the SDF analysis technique yields good results for high resource loads, whereas SymTA/S shows superior analysis quality for low resource loads. The high overestimation on the jitter using SDF analysis for low resource load is caused by the construction of the token placement used to obtain the highest possible output load (see Sec. 4.3), as the algorithm does not take constraining inputs into account.

Secondly, we present analysis results for the complete setup shown in Fig. 1. For input traffic parameters as shown in Table 2(a), we compute output parameters as shown in Table 2(b) using the approach presented in this paper.

6 Conclusion

We have shown in this paper how the well established theory of SDF graph analysis that is widely used in signal processing optimization can be included in a compositional performance analysis. We have demonstrated the superior analysis resulting from this combination. This was possible by reasoning about the input and the output behavior of SDF graphs and satisfying the event model interface of the compositional analysis. To allow reasoning about the system's end-to-end behaviour, we also computed path latencies.

(a) Input Event Models

Source	$\mathcal{P}$	$\mathcal{J}$	d^{min}
In1	1700	0	0
In2	750	0	0

(b) Output Event Models

Sink	$\mathcal{P}$	$\mathcal{J}$	d^{min}
Out1	1700	2590	810
Out2	750	495	0

Table 2. Input and Output Event Models

Using this approach, we achieve three goals: complex dataflow analysis can be decomposed to achieve better computability, dataflow graphs may be embedded into compositional performance analysis to increase precision, and a mix of both models may be used to extend the scope of setups for which performance predictions are possible.

References

[1] F. Bacelli, G. Cohen, G. J. Olsder, and J.-P. Quadrat. *Synchronization and Linearity*. John Wiley & Sons, Inc., 1992.

[2] W.-T. Chang, S. Ha, and E. A. Lee. Heterogeneous simulation - mixing discrete-event models with dataflow. *J. VLSI Signal Process. Syst.*, 15(1-2):127–144, 1997.

[3] A. Dasdan and R. Gupta. Faster maximum and minimum mean cycle alogrithms for system performance analysis. *IEEE Transactions on Computer-Aided Design*, 17(10):889–899, October 1998.

[4] A. Ghamarian, M. Geilen, S. Stuijk, T. Basten, A. Moonen, M. Bekooij, B. Theelen, and M. Mousavi. Throughput analysis of synchronous data flow graphs. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[5] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis - the symta/s approach. In *IEE Proceedings Computers and Digital Techniques*, 2005.

[6] S. Künzli, F. Poletti, L. Benini, and L. Thiele. Combining simulation and formal methods for system-level performance analysis. In *Proc. Design, Automation and Test in Europe*, 2006.

[7] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9), 1987.

[8] A. Moonen, M. Bekooij, and J. van Meerbergen. Timing analysis model for network based multiprocessor systems. In *Proceedings of the 5th progress symposium on embedded systems*, pages 122–130, Octobre 2004.

[9] K. Richter. *Compositional Scheduling Analysis Using Standard Event Models*. PhD thesis, Technical University of Braunschweig, 2004.

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

[11] S. Stein. Embedding a data flow-based real-time performance analysis into an event-model based multiprocessor evaluation technique. Master's thesis, Technical University of Braunschweig, January 2006.

[12] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systeme. In *Proc. IEEE International Conference on Circuits and Systems*, 2000.