Hybrid Latency Minimization Approach using Model Checking and Dataflow Analysis

Guus Kuiper §
g.kuiper@utwente.nl

Philip S. Kurtin §
philip.kurtin@utwente.nl

Marco J.G. Bekooij ¶ §
marco.bekooij@nxp.com

§ University of Twente, Enschede, The Netherlands ¶ NXP Semiconductors, Eindhoven, The Netherlands

ABSTRACT

Bounding the latency of real-time multiprocessor applications is crucial for safety-critical systems. Several approximative analysis approaches exist that can efficiently analyze the latency. However, these approaches produce pessimistic latency results and do not exploit buffer sizing nor exploit additional sequence constraints to reduce the latency. More accurate latency analysis results can be obtained using model checking of timed-automata, however, at the cost of a typically excessive run-time.

This paper presents a latency analysis approach for cyclic task graphs using model checking of timed automata of which the run-time is reduced. The approach is applicable for systems in which tasks are executed on shared processors using a Fixed Priority Pre-emptive (FPP) scheduling policy. The reduction in run-time is achieved by pruning the search space of options that need to be analyzed using the model checker by making use of approximative dataflow analysis techniques. The approach exploits dimensioning of buffers to minimize interference and latency. Moreover, sequence constraints are introduced and automatically adapted in order to minimize the latency of the task graph.

A WLAN 802.11p transceiver application is used in the case study to compare this hybrid analysis approach to a state-of-the-art approximation based approach that uses iterative buffer sizing. Using our approach, the analyzed latency decreased from $17\,\mu$s to $15\,\mu$s at the cost of a run-time of 23 minutes instead of a fraction of a second.

CCS Concepts

•Software and its engineering → Formal software verification;

1. INTRODUCTION

Safety-critical applications often impose a strict latency requirement on their implementation. Automated braking systems, for example, require their inputs to be processed within a certain latency to be able to avoid collisions. In this paper we assume that these applications are executed on a multiprocessor system. Each application is described as a task graph, in which the tasks are executed on shared processors. A FPP scheduling policy is often used to share the processor between all tasks assigned to it. Formal analysis techniques must be used to verify at design time whether the latency requirements of safety critical applications are met.

Cyclic dependencies in such an application complicate analysis. However, these dependencies can also have beneficial effects on latency. These cyclic dependencies can for example be a result of: feedback loops, a static task execution order, clustering of tasks [5] or bounded size First-In-First-Out (FIFO) buffers [22]. As a result of these cyclic dependencies, tasks on these cycles can only interfere a bounded number of times with each other. The maximum number of times tasks can interfere with each other can for example be controlled by the size of the buffer between tasks [22]. Lower interference between tasks can have a positive effect on the maximum latency. But, as we will show in Section 7, minimal buffer sizes, given a throughput constraint imposed by a periodic source, do not always result in the minimal latency.

Buffer sizing cannot be used to control the interference between tasks if these tasks do not communicate. Moreover, sizing of buffers cannot delay the start of a consuming task. To provide more control over the schedule freedom we introduce sequence constraints with initial tokens between tasks. The interference between tasks, and as a result the maximum latency of the task graph, can be reduced by carefully controlling the number of initial tokens. We will show that in some cases a negative number of initial tokens is required for achieving the minimum latency. The number of initial tokens on sequence edges can be derived with a similar algorithm as is used for buffer sizing.

Several approximative analysis approaches exist that can calculate the latency for real-time multiprocessor applications, including the effects of FPP scheduling. However, well known approaches like MAST [7] and SymTA/S [10] are not able to analyze arbitrary cyclic dependencies. Therefore, these approaches are not able to optimize the latency by exploiting cyclic dependencies. The iterative dataflow analysis approach in [15] is able to handle arbitrary cyclic dependencies and uses a buffer sizing technique, which can result in a latency reduction. This approach does not consider all possible buffer sizes otherwise monotonicity and convergence of the used analysis flow cannot be guaranteed.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SCOPES '17, June 12-14, 2017, Sankt Goar, Germany

© 2017 ACM. ISBN 978-1-4503-5039-6/17/06...$15.00

DOI: http://dx.doi.org/10.1145/3078659.3078665

Dataflow analysis techniques have shown to be suitable for the analysis of cyclic applications. Processor sharing, however, cannot be modeled explicitly and can only be included by deriving an over-approximation bound on the interference of tasks. There are techniques to prevent interference between tasks that produce more accurate analysis results by making tasks mutually exclusive [14]. However, making tasks mutually exclusive does not always result in the minimum latency. As shown in [13], exact latency analysis for dataflow graphs without pre-emptive scheduling is possible using timed automata if the execution times are natural numbers.

Timed automata are a popular formalism for modeling and analyzing real-time systems. Although latency analysis is possible, there is no computationally efficient build-in support for optimizing of costs like, buffer sizes, or the introduction of additional sequence constraints, to reduce latency. The problem of finding the minimum latency in case of cyclic dependencies is non-monotone, therefore all configurations of buffer sizes have to be analyzed. Moreover, without a combination of timed automata and an approximate analysis technique to upper bound and reduce the number of configurations, many of these configurations need to be analyzed leading to a high run-time.

This paper presents a latency minimization approach with a reduced run-time for cyclic task graphs executed on shared multiprocessor systems using FPP task scheduling. The approach identifies the configuration of buffer capacities and sequence constraints that result in the minimum latency. The approach uses approximative, but computationally efficient, timed-dataflow analysis techniques for removing all configurations from the search space that cannot result in the lowest latency since these will deadlock, won't reach the throughput of the periodic source, or impose redundant constraints. To find the configuration with the lowest latency the remaining options are evaluated using a model checking approach.

This paper is structured as follows. In Section 2, we discuss related latency analysis approaches for task graphs using FPP scheduling. The basic idea behind our hybrid analysis approach is presented in Section 3. In Section 4, timed automata are introduced, which we use to create a model of task graphs. Model checking is performed on these models to analyze latency. The dataflow model is introduced in Section 5, which is used in analysis techniques that reduce the number of configurations that need to be analyzed. Both these timed automata and dataflow-based analysis techniques are combined in our analysis flow as described in Section 6. In Section 7, we introduce sequence constraints to reduce latency. The case study is presented in Section 8. We state our conclusions in Section 9.

2. RELATED WORK

In this section we present work related to latency analysis of task graphs where tasks are scheduled using FPP scheduling. We first discuss related approximative analysis approaches, which do not perform buffer sizing to reduce latency. Then we discuss analysis approaches that use timed automata and decidability of scheduling problems using timed automata. Finally, we discuss an approach that does perform buffer sizing.

The SymTA/S approach [10], is one of the techniques which overestimates interference between tasks in order to analyze latency. The approach, however, is not able to handle arbitrary cyclic dependencies. MAST [7] is another approach that can derive a more accurate characterization of the interference. It is limited to the analysis of acyclic graphs. The MPA-RTC [20] analysis approach has been extended to support either cyclic data dependencies [21], or cyclic resource dependencies [11]. The combination of cyclic resource and data dependencies is not supported. Neither of these approaches use buffer sizing techniques to reduce latency, nor introduce additional sequence constraints to reduce latency.

Timed automata based analysis approaches can be used to determine the latency of a task graph. A number of these approaches only considers a fixed (worst-case) execution time of tasks [9, 17, 18]. This ignores additional interference between tasks that can occur when tasks finish earlier. The TIMES-tool [1, 6] does consider Best-Case Execution Times (BCETs) and Worst-Case Execution Times (WCETs). We use their method of summing up the time needed to finish all released tasks to determine when a task finishes its own execution, which is described in more detail in Section 4. However, TIMES only considers the single processor case. The multiprocessor case is considered in [2, 3], but these approaches are limited to acyclic task graphs. Moreover, none of these timed automata based approaches exploit buffer sizing of blocking buffers to reduce latency.

Decidability of scheduling problems using timed automata is discussed in [12]. This paper shows that the problem of deciding whether the deadlines are met of tasks in a task graph that is scheduled using fixed priority scheduling on a single processor, is in general undecidable. Therefore also the exact latency of an arbitrary task graph on shared resources can not be computed. More precisely the problem to verify whether the tasks meet their deadlines is undecidable if the following three conditions hold: 1) the execution times of the tasks are characterized by an upper and lower bound on a continuous time interval, 2) task can announce there completion time to other tasks at every point in continuous time, and 3) each task can preempt another task at any point in time, i.e. not only at clock cycle boundaries. The problem is not a decision problem anymore if the result of the analysis can be besides the yes/no answer also the answer that may-be the tasks meet their deadlines. Over-approximation during analysis introduces this third option and this is what has been used in [1, 6] and is also used in this paper. The approach in [3] makes use of stopwatch automata which are in general undecidable. In Uppaal 4.1 techniques are introduced that apply over-approximation during analysis in case stopwatches are applied in the timed-automata model. This over-approximation has as a positive side-effect that the run-time is reduced. We introduce in this paper also a timed-automata model in which stopwatches are used. This enables a comparison with the case that a timed-automata model without stopwatches is used. In [16] it is not assumed that tasks can finish at every point in time. This makes the schedulability of the decision problem decidable but results in an impractically large automata model if it is considered that tasks can be preempted at every clock cycle boundary.

The approach in [15] minimizes buffer sizes given a throughput constraint. It, however, does not minimize the latency

at the same time. We do make use of this approach in our hybrid analysis flow to find bounds on buffer capacities, by analyzing the buffers as non-blocking buffers. This approach over-approximates interference. Moreover, a small generalization had to be made to be able to compute the number of initial tokens on sequence constraints.

3. BASIC IDEA

The idea behind our analysis approach is presented in this section. We start by showing the relation between a task graph containing a blocking buffer and a dataflow model of it, where cyclic dependencies are present. Then, we show how these cyclic dependencies can influence interference between tasks. Finally, we indicate how we can minimize latency using these cyclic dependencies.

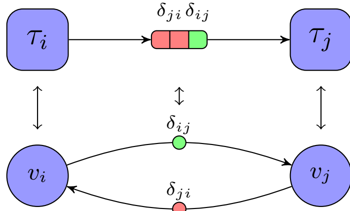


Figure 1: One-to-one relation between a task graph containing a blocking buffer and a dataflow graph.

The dependencies in a task graph have a one-to-one relation with the edges in a dataflow graph. This is illustrated in Figure 1, where two tasks, τ_i and τ_j, communicate using a blocking buffer. Initially this buffer contains one full container δ_{ij} and two empty containers δ_{ji}. A task can only execute and write to such a buffer after it first obtained an empty container. After its execution, it produces a full container, which can be read by the consuming task. In the dataflow graph, the initially full containers of the buffer are represented as tokens on the edge from actor v_i to v_j, which correspond to the tasks, whereas the tokens on the edge from v_j to v_i represent empty containers.

Scheduling freedom is affected by these cyclic dependencies that model blocking buffers. Based on these cyclic dependencies we can derive an upper bound on the number of times tasks can be pre-empted using the FPP scheduling policy. The number of times τ_i can be pre-empted by τ_j is therefore bound by $\delta_{ij} + \delta_{ji}$. Therefore we can conclude that interference between tasks can be controlled by choosing the number of containers in the buffer, e.g. by using buffer sizing techniques.

The relation between buffer sizes and latency for a graph is non-monotone [22]. Therefore, in order to obtain the minimum latency for a task graph, we have to verify latency for all possible buffer sizes of all buffer, which we call configurations in this paper. In order to obtain latency results for this unstructured analysis problem, model checking is performed. An upper bound on the size of each buffer can be calculated using approximative analysis techniques. The upper bound represents the situation where the schedule of tasks is not influenced by the size of buffers because there are always sufficient empty containers. As a result the write of a buffer will never be blocked and thus delayed because there is always space in the buffer. The lower bound on the buffer size is determined by the number of full containers. In case it is equal to one a fixed execution order of the tasks is enforced. Such an order is called a static-order schedule. Timed automata of the task graph are generated for all possible buffer sizes between these computed lower and upper bounds. These timed automata are then analyzed using Uppaal to find a configuration with the minimum latency.

4. TIMED AUTOMATA

A template-based timed automata model is constructed in this section to allow different configurations of a task graph to be verified automatically using Uppaal. We first present the extended timed automata model which is used in Uppaal. Using these extended timed automata, we define parameterized templates for the different components in a tasks graph[1]. The maximum latency of a task graph is then verified using Uppaal for a network of timed automaton instances.

An extended timed automaton, as used in Uppaal, is a directed graph $\mathcal{A} = (L, Act, C, E_a, Inv, l^0)$ where L is a set of locations, Act a finite set of actions, C a set of clocks, $E_a \subseteq L \times Act \times B(C) \times 2^C \times L$ is a set of edges, $Inv : L \rightarrow B(C)$ assigns an invariant to each location and l^0 is the initial location. The invariant of a location expresses an upper bound $\hat{c} \in \mathbb{N}$ on a clock $c \in C$. A location $l \in L$ can be urgent, meaning that no time is allowed to pass when an automaton is in l. A committed location defines that an automaton must leave the location immediately. These committed locations are used to create atomic sequences.

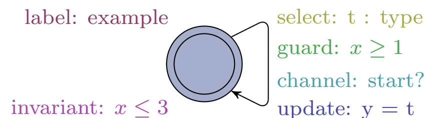
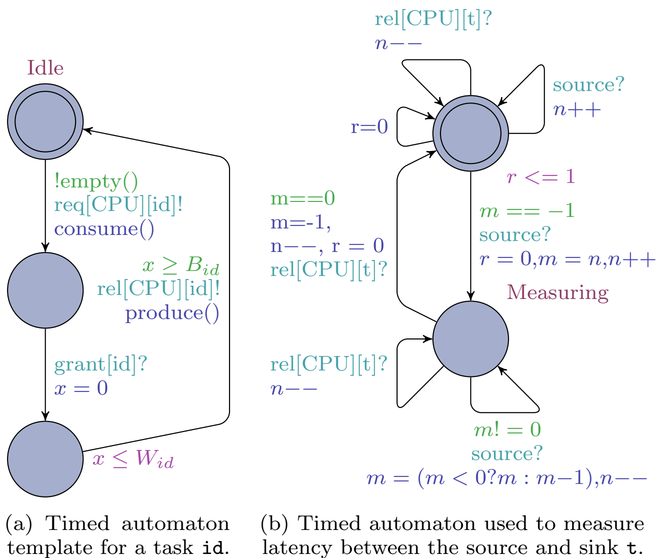


Figure 2: Extended timed automaton annotations.

An edge in the automaton can have the following annotations as shown in Figure 2: *selects*, *guards*, *synchronizations* and *updates*. A *select* non-deterministically chooses a value in a bounded range after a transition, which can be used as variable in the other annotations. A *guard* specifies when a transition over the edge is allowed by means of a boolean expression or integer bounds on clocks. The *synchronizations* annotation can be used to define synchronization of transitions in different automata. Finally, an *update* states integer expressions of which the outcomes are assigned to variables after a transition on the edge is taken.

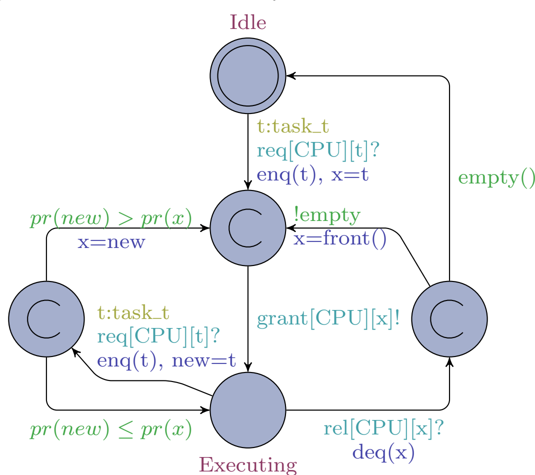
A network of timed automata can be defined as a composition of extended timed automata, which can synchronize with each other. Synchronization between these automata can be implemented with channels or global variables. A channel c has an emitting edge labeled *c!* and potentially has multiple receiving edges labeled *c?*. These receiving edges block until an event is received. Only when a transition on both sides of the channel is enabled, a transition will occur simultaneously across both edges. For an urgent channel the transitions cannot be delayed and must occur immediately once they are enabled. A broadcast channel can have multiple receiving edges simultaneously synchronizing

[1]Available at https://github.com/gkuiper/UppaalWLAN



(a) Timed automaton template for a task `id`.

(b) Timed automaton used to measure latency between the source and sink `t`.



(c) Timed automaton template for a processor `CPU` using FPP scheduling.

Figure 3: Timed automata templates for different components in a task graph.

to one emitting edge. However, a transition across the emitting edge may occur without one of the receiving edges being enabled. A transition over an edge can also update a global variable. A transition over an edge in another automaton can be triggered when it contains a guard function returning a boolean which depends on the value of that global variable.

Based on these extended timed automata we will now define a template for tasks, schedulers, and FIFO buffers. Given these templates, a network of timed automata is created for a specific configuration of a task graph where the buffer sizes are configured and a number of tasks and processors are instantiated. Each configuration will therefore result in a unique network of timed automata.

4.1 FIFO buffer

A FIFO buffer is modeled in extended timed automata using global variables and access functions to manipulate and check these variables [14]. Each buffer is represented by two edges, each containing the producer task, consumer task, and a number of tokens on the edge. The number of tokens is initialized with the number of full or empty containers in a buffer. The function `empty()` is used to check if there are insufficient tokens, e.g. $tokens < 1$. Tokens are produced using `produce()`, which increments the number of tokens by one. The number of tokens is decremented by one using `consume()`.

4.2 Task template

The timed automaton of a task consists of three locations as shown in Figure 3a. The transition from the initial location, Idle, is guarded by the expression which checks if there are sufficient tokens on *all* its incoming edges. The urgent broadcast channel req is used to signal the processor that the task is ready to execute and tokens are consumed from all incoming edges using *consume()*. In the second location, time passes until execution on the processor is granted using the urgent broadcast channel grant. During the transition to the 3rd location, the clock x is reset. This clock represents the execution time of the task. This location can be left when the clock is between B_{id} and W_{id} which are equal to the BCET and WCET. During the transition to location Idle tokens are produced and the processor is signaled about the completion of the task using the broadcast channel rel.

Two options are implemented to incorporate the inference caused by pre-emptions of tasks with a higher priority. One option uses over-approximations of the finish times to guarantee decidability of the scheduling problem, the other option uses stopwatches.

4.2.1 Over-approximating pre-emptions

Pre-emptions can be over-approximated by deriving a lower and upper bound on the completion time of higher priority tasks. For each pre-emption the bounds of lower priority tasks, B_{id} and W_{id} in Figure 3a, are incremented respectively with the BCET and WCET of the task which triggered the pre-emption. This is an over-approximation because their are executions of the higher priority task that are smaller than its WCET.

4.2.2 Stopwatches

Using stopwatches a more intuitive task template can be constructed in which the clock tracking the execution time is paused when a task is pre-empted. The processor must set a global variable `runs[CPU]` to indicate which task is currently executing on a processor. The invariant of the 3rd location in Figure 3a must be extended to also set the derivative of clock x to either 0 or 1 depending on the tasks that is executing, e.g. `x' == (runs[CPU]==id)`. The clock is stopped if `x' == 0`. The global variable `runs[CPU]` is set to the *id* of the task that is granted access to the processor and is reset when the task releases the processor.

4.3 Processor template

A processor template for the FPP scheduling policy is constructed using a task queue. This queue contains all tasks that have issued a request. For FPP scheduling, the queue is sorted based on the priority of the tasks. The timed automaton consists of two locations where time can pass, Idle and Executing, and three committed locations used to create atomic sequences where time cannot advance. This timed automaton is shown in Figure 3c. From the Idle and Executing location, a transition occurs when a task issues a request via the req channel. In that case, the new task is

enqueued, enq(t). When the transition is taken from Idle the queue was empty and the task is immediately granted access to the processor using the channel grant. Otherwise, the priority of the currently executing task and the new task are compared. Based on the outcome, the current task can resume execution or is pre-empted and the new task is granted access to the processor. After a task finished its execution, the processor receives a signal on the rel channel and as a result, removes that task from the queue. The task in the front of the queue with the highest priority is then granted access, or the processor idles if the queue is empty.

4.4 Verification

The latency in a network of timed automata is measured using a separate timed automaton. The automaton as shown in Figure 3b measures the latency between an event produced by the source from channel source and the release event of the task producing the output for the sink, using the channel rel. The source, which produces events, is characterized by a period and jitter. Since multiple source events can be produced before a release event is produced that is a result of the first source event, a non-deterministic selection is made of an event that is tracked. This non-deterministic selection results in that UPPAAL will check all source events. Variable n tracks the number of outstanding events from the source and is incremented for each event from the source. This variable is copied to variable m when a source event has occured and the transition to the location Measuring is made. Variable m tracks the number of remaining release events before the event corresponding to the source event is found. Release events belonging to previous source events are skipped and decrement m. Only when both $m == 0$ and a release event occurs, the matching release event is found such that the latency measurement is stopped by returning to the initial location. In the initial location, the clock r is reset every time unit since no measurement is taking place. This is implemented using the invariant `r <= 1` and the edge with the update `r = 0`. The maximum latency can then be obtained by querying the suprema of clock r in the location Measuring: `sup{Measuring}: r`.

5. TIMED-DATAFLOW

Computationally efficient approximative analysis of task graphs can be performed using timed-dataflow analysis techniques. In this section we will first formally introduce the Homogeneous Synchronous Dataflow (HSDF) model. Based on this dataflow model we derive techniques to prune the number of configurations that need to be analyzed in our approach using timed automata. Finally, we describe dataflow-based approximative analysis method that we use to derive upper bounds on buffer capacities.

An HSDF graph is a directed graph $G = (V, E, \delta, \rho)$ that consists of a set of actors V connected by a set of directed edges E. An actor $v_i \in V$ communicates with another actor v_j by producing tokens on an edge $e_{ij} \in E$. Initially there are $\delta_{ij} \in \mathbb{N}_0$ tokens on an edge. An HSDF actor v_i is enabled to fire its i-th iteration if there is at least one token on all its incoming edges. When an actor fires, one token is consumed from each of its incoming edges and one token is produced on each outgoing edge of the actor. The tokens are produced exactly $\rho_x(i) \in [\check{\rho}_x, \hat{\rho}_x]$ after the enabling of the actor. The firing duration $\rho_x(i)$ in each iteration i is chosen non-deterministically from the interval that is bounded by the minimum firing duration $\check{\rho}_x$ and maximum firing duration $\hat{\rho}_x$ of v_x.

The dependencies of a task graph can directly be translated into an HSDF graph as discussed in Section 3. However, the effect of FPP scheduling cannot be modeled directly in HSDF graphs because HSDF graphs do not allow modeling of choice. Therefore, bounds on the response times of tasks are used as firing durations of the corresponding actor. As a consequence accurate analysis latency results cannot be obtained directly but these approximations can be used to prune the number of configurations that need to be analyzed using UPPAAL.

5.1 Deadlock

By definition, a HSDF graph deadlocks if it contains a simple cycle without initial tokens. We can therefore prune, i.e. discard, all configurations with simple cycles without initial tokens. This, is especially useful after introducing additional constraints as presented in Section 7. UPPAAL can also verify the absence of deadlock, however, less computationally efficient.

5.2 Minimum guaranteed throughput

Another method to prune the number of configurations is by considering the throughput constraint imposed by the source. The guaranteed throughput of HSDF graph must be greater than the throughput of the source to prevent the source from blocking as a result of a full buffer. We cannot determine the throughput of a dataflow graph accurately when FPP scheduling is involved. However, a lower bound on the guaranteed throughput can be determined by setting the firing duration of all actors to the WCET of the corresponding tasks, ignoring any interference. A computationally efficient Maximum Cycle Ratio (MCR) algorithm [19] can then be used to calculate the MCR, which corresponds to the inverse of the minimum guaranteed throughput.

5.3 Approximative dataflow analysis

The approximative dataflow analysis method we use is based on execution intervals of tasks [15]. The execution interval $\mathcal{I}_i$ consists of the earliest possible start time and the latest possible finish time of a task τ_i. These intervals are determined from a best-case dataflow model using BCETs, and a worst-case dataflow model using WCETs and upper bounds on task interference.

The analysis flow is shown in Figure 4. Initially, the execution intervals are determined without considering interference as shown in step 1 of the flow. Initial buffer sizes are chosen such that the graph is deadlock free. In step 2, an upper bound on the possible interference between tasks is calculated given the execution intervals. The precedence constraints originating from the buffers in a task graph are included in this interference calculation. The execution intervals are extended in step 3 using the calculated interference. Buffer sizes for iteration $k \in \mathbb{N}$ are calculated based on the latest finish time of the consuming task τ_j, $\hat{\mathcal{I}}_j$, and earliest start of the producing task τ_i, $\check{\mathcal{I}}_i$:

$$\delta_{ji}^k = \max\left(\left\lceil \frac{\hat{\mathcal{I}}_j - \check{\mathcal{I}}_i}{P_j} \right\rceil, \delta_{ji}^{k-1}\right) \quad (1)$$

where δ_{ji}^k is the number of initial tokens on e_{ji} in iteration

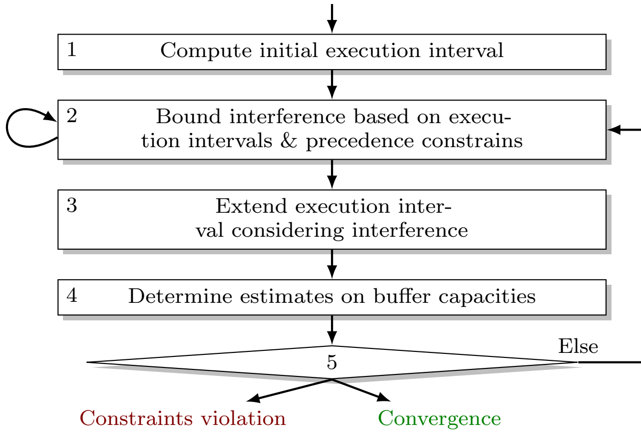


Figure 4: Analysis flow of the approximative dataflow analysis approach.

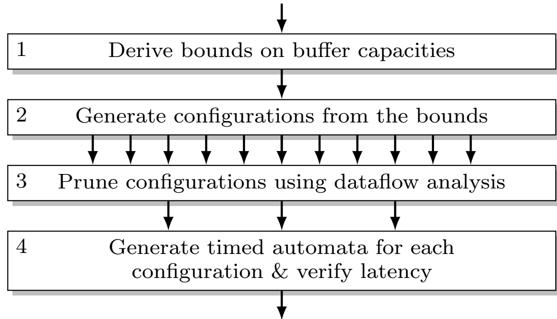


Figure 5: Hybrid analysis flow.

k and P_j is the period of the consuming task. The buffer sizes are not allowed to decrease compared to the previous iteration in order to guarantee termination of the analysis flow. Step 2 to 4 form an iteration of the analysis flow, which is repeated until a constraint violation occurs or the execution intervals and buffer sizes have converged, i.e. stay the same.

Since buffer capacities are determined in every iteration of this analysis flow, we refer to this method as iterative buffer sizing. Note that we cannot guarantee that the configuration of buffer sizes leading to the minimal latency of a task graph is considered using iterative buffer sizing.

The analysis flow can also analyze buffers without blocking write, i.e. non-blocking buffers. In that case, the buffer sizes are calculated such that there always is at least one empty container available for a task that is enabled, to prevent buffer overflows. Analyzing blocking buffers as non-blocking buffers allows derivation of an upper bound on their capacity.

6. HYBRID ANALYSIS

Our hybrid analysis approach, which combines analysis techniques from timed-dataflow and timed automata, will be described in this section. The hybrid analysis flow is shown in Figure 5, which will be described step-by-step. The flow uses a task graph as an input containing buffers, tasks, and a task to processor assignment.

In the first step, a lower and upper bound is derived on the capacity of all buffers. The upper bound is calculated using the approximative analysis technique as described in Section 5. This upper bound is determined by making each buffer non-blocking. The lower bound is set to the maximum of 1, which prevents deadlock, and the number of initially full containers in the buffer.

Once these bounds are determined, configurations of the task graph are generated in step 2. Since the latency is non-monotone in the buffer capacities, we have to consider all possible sizes of each buffer within these bounds. The set of configurations therefore consists of the product of all sizes of all buffers.

Many configurations are generated for which we can guarantee, using dataflow analysis techniques, that these will not result in the minimal latency. Step 3 performs pruning of the configurations using the deadlock and minimum guaranteed throughput techniques described in Section 5.

In the final step, a network of timed automata is created for each remaining configuration. As described in Section 4, a timed automaton template is initiated for all buffers, tasks and processors. The sizes of the buffers are fixed as specified in the configuration. The model checker UPPAAL is used to verify the maximum latency of the network of timed automata. Once all configurations are verified, the one resulting in the minimum latency is selected to finish the analysis.

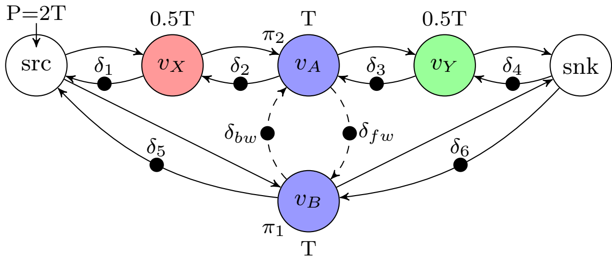
In the next section we discuss the introduction of additional sequence constraints. The same hybrid analysis flow can be used for these constraints, since these constraints can be seen as a generalization of the constraints imposed by buffers.

7. SEQUENCE CONSTRAINTS

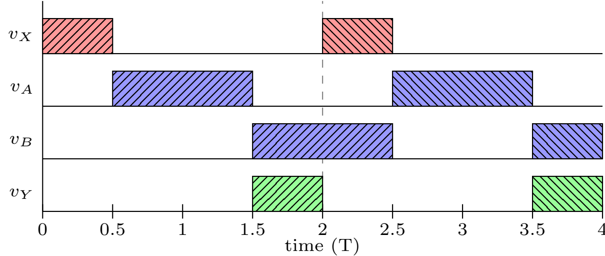
In this section we present a method that potentially reduces latency by introducing additional sequence constraints. We introduce pairs of these sequence constraints as cyclic dependencies between two tasks to have direct control over the maximum interference one of these tasks can cause on the other. This construct of sequence constraints can be seen as a generalization of a blocking buffer, with as important difference that it does not store data.

In general, buffers only allow their capacity to be changed in case it does not result in a change of the functional behavior of an application. The number of initially full containers in a buffer is fixed and therefore there is only control over the number of initially empty containers. As a result, changing the buffer capacity will only change the temporal behavior of an application and not the functional behavior (unless it deadlocks). The sequence constraints can be seen as a generalized buffer, it not only allows control over the number of initially empty, but also over the number of initially full containers. Adding these sequence constraints does not influence the functional behavior of an application because it only affects the schedule freedom and does not result in transfer of data.

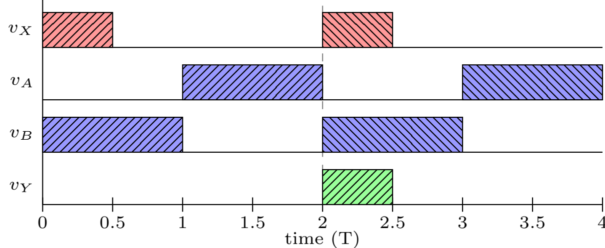
Formally, a sequence constraint between τ_A and τ_B contains a number of containers $\delta_{AB_{fw}}$ on the forward edge e_{AB}, and a number of containers $\delta_{AB_{bw}}$ on the backward edge e_{BA}. Let $s_A(i)$ be start of the i'th ($i \in \mathbb{N}$) execution of τ_A, $f_A(i)$ the finish of the i'th execution of τ_A and logically $f_A(i) \geq s_A(i)$. The sequence constraints introduce the following constraints: $s_B(i) >= f_A(i - \delta_{AB_{fw}})$ and $s_A(i) >= f_B(i - \delta_{AB_{bw}})$. Notice that a deadlock will occur when $\delta_{AB_{fw}} + \delta_{AB_{bw}} \leq 0$. In case $\delta_{AB_{fw}} + \delta_{AB_{bw}} = 1$, only one execution order is enforced between τ_A and τ_B, which



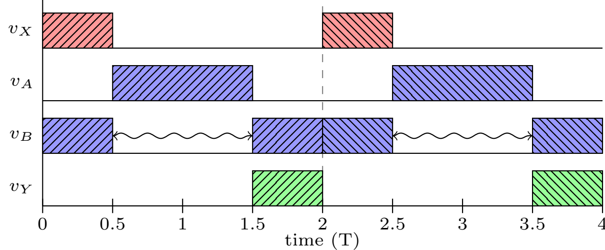
(a) Dataflow graph where a static execution order of v_A and v_B increases the end-to-end latency.



(b) Schedule of Figure 6 for which $\delta_{fw} = 0, \delta_{bw} = 1$.



(c) Schedule of Figure 6 for which $\delta_{fw} = 1, \delta_{bw} = 0$.



(d) Schedule of Figure 6 for which $\delta_{fw} = 1, \delta_{bw} = 1$.

Figure 6: Example where a static execution order of v_A and v_B increases the end-to-end latency.

therefore is a static execution order.

A one-to-one translation is possible from a sequence constraint to a corresponding dataflow model. The forward and backward containers of a sequence constraint are equivalent to initial tokens on the corresponding edge. An example of a dataflow model containing sequence constraints is shown in Figure 6. The example consists of four tasks; τ_A, τ_B, τ_X and τ_Y. τ_A and τ_B share a processor and sequence constraints are therefore inserted between τ_A and τ_B. These tasks are converted to actors in the dataflow model. The edges corresponding to the sequence constraints are shown as dashed lines in the graph.

Since the sequence constraints can directly be expressed in a dataflow model, existing analysis techniques can be applied to it that determine the required number of tokens given a throughput constraint. The difference with buffer sizing for the number of empty containers is that, for each sequence constraint, the number of containers on two edges can freely be chosen. The approximative timed-dataflow approach in [15] is therefore extended to derive an upper bound on the number of initial tokens on both the forward as the backward edge using the existing buffer sizing algorithm. The hybrid analysis, as presented in Section 6, is therefore also applicable for graphs containing sequence constraints.

We will use the examples in Figure 6 to illustrate that enforcing a static execution order between τ_A and τ_B by using sequence constraints will lead to a higher latency than when τ_A (with a high priority) can pre-empt τ_B (low priority). A static order schedule between these tasks can be created where τ_A must execute before τ_B by selecting $\delta_{AB_{fw}} = 0, \delta_{AB_{bw}} = 1$ as shown in the schedule in Figure 6b. The other option is to execute τ_B first, $\delta_{AB_{fw}} = 1, \delta_{AB_{bw}} = 0$ as shown in the schedule in Figure 6c. Both options result in a latency from source to sink of 2.5T as computed by our hybrid analysis approach. However, selection of $\delta_{AB_{fw}} = 1, \delta_{AB_{bw}} = 1$, as shown in the schedule in Figure 6d, results in a latency of 2T. Therefore, we can conclude that enforcing a static execution order between tasks mapped to the same processor does not always result in the minimum latency.

7.1 Negative tokens

Recently the use of negative tokens has been introduced in dataflow models [4, 8]. A negative number of tokens delays the enabling of an actor since the number of tokens on an edge must be positive before the consuming actor is enabled. A negative number of tokens can be used in sequence constraints to model that there are several initial executions of a task before a static execution order between two tasks starts.

Figure 7 shows an example of a dataflow graph, for which enforcing a static execution order, without using negative tokens, will result in a throughput violation. A static order is enforced by setting $\delta_{fw} = 0, \delta_{bw} = 1$. In that case there are insufficient tokens on cycle $v_A \rightarrow v_B \rightarrow v_C \rightarrow v_A$ to keep up with the periodic source. When introducing a negative token, $\delta_{fw} = -1, \delta_{bw} = 2$, the throughput of the source will be achieved, while there is a static execution order after v_A is execute twice.

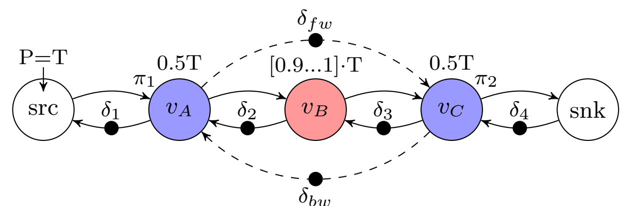


Figure 7: Dataflow graph where a negative number of tokens on δ_{fw} between v_A and v_C can be useful.

For this example, creating a static order with negative tokens results in a latency of 2T. Whereas, without negative tokens, the latency will be 3T. The range of the firing duration of v_B can result in v_C pre-empting v_A, which leads to this increased latency. The approximative analysis method [15] is not able to compute a feasible result since its analysis did not converge.

7.2 Redundant constraints

Sequence constraints introduce additional constraints in an already connected dataflow graph of an application. Therefore, adding sequence constraints can lead to redundant constraints, more configurations and an increased in run-time. We now present two techniques to reduce this redundancy.

We will exploit that within one application graph, a path is required from the source to all tasks without tokens. In a dataflow model of that graph, there are tokens on the edges in the opposite direction to represent buffers with a minimum of one empty container. This connected graph allows us to create a token distance matrix, containing the minimum number of tokens on the path $\mathcal{P}(i, j)$ between all actors i, j. For each configuration with buffer sizes and specific number of tokens on the forward and backward edge of the sequence constraints, this matrix can be constructed.

First of all, the sequence constraints are always redundant when these sequence constraints are inserted between two tasks, τ_A and τ_B, that are already connect by a buffer. Such sequence constraints can safely be removed to reduce the number of configurations. An example is shown in Figure 8, where the sequence constraints between v_A and v_B can be removed to reduce the number of configurations.

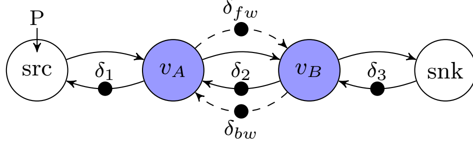


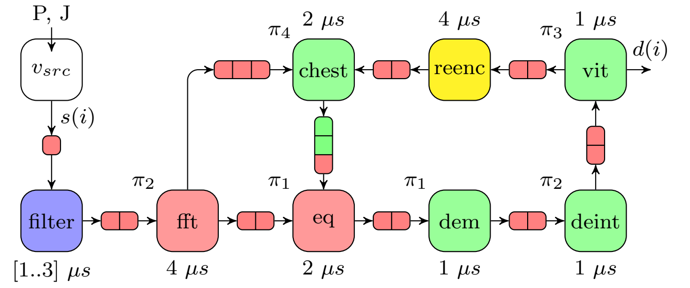
Figure 8: Dataflow graph where the sequence constraints between v_A and v_B is redundant to the constraints of the buffer between the actors.

Secondly, sequence constraints impose redundant constraints if: $\delta_{fw_{AB}} > \mathcal{P}(from, to)$ or $\delta_{bw_{AB}} > \mathcal{P}(to, from)$. Configurations where this equation holds can be skipped without the possibly of missing a configuration, which leads to the minimal latency. In the example shown in Figure 7, configurations where $\delta_{bw} > \delta_2 + \delta_3$ can be skipped.

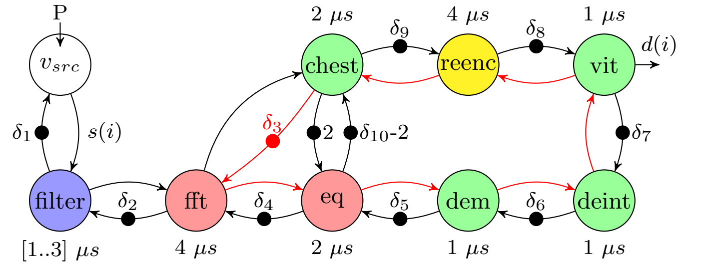
8. CASE STUDY

In this section we will compare our timed automata-based analysis approach to a state-of-the-art approximative analysis approach for a WLAN 802.11p transceiver application which contains cyclic dependencies due to a feedback loop in the algorithm.

The application consists of eight tasks that are each mapped to one of four processors using a FPP scheduling policy. Figure 9a shows the task graph of this application, where the colors of the nodes represent a mapping to a processor. The priority of tasks is indicated next to them, as π_i, where π_1 is the lowest possible priority. The figure also shows the WCET of all tasks or the range [BCET..WCET] for tasks without constant execution times. The color of the containers in the buffers connecting the tasks indicates if they are initially empty (red) or full (green). The number of containers in each buffer in Figure 9a shows the size of the buffer, which does not influence the schedule of the tasks as determined by an approximate analysis approach. The sum of these buffer capacities is 21. In this example, the source has a period of $10\,\mu s$ and a jitter of $5\,\mu s$. For this application, we want to



(a) Task graph of a WLAN 802.11p transceiver application.



(b) Dataflow model of Figure 9a without including interference.

Figure 9: Task graph and equivalent dataflow model of a WLAN 802.11p transceiver application.

minimize the maximum latency between the i'th start of the source, v_{src}, and the corresponding i'th production of the Viterbi task, *vit*.

The approximate analysis approach makes use of the fact that lower buffer capacities can result in a lower end-to-end latency [15]. Buffer sizing is therefore performed iteratively inside the analysis algorithm and sizing of buffers is not deferred until the analysis of response times of all tasks is computed. In order for the analysis flow to guarantee convergence, the iterative buffer sizing step increases the buffer sizes monotonically. Therefore, this buffer sizing technique does not guarantee that all possible buffer sizes are considered.

Our timed automata-based approach can offer more accurate latency analysis given buffer capacities. We analyze all possible buffer sizes, thereby guaranteeing the buffer size configuration that leads to the minimum latency to be present in the analyzed set of configurations. However, many of these configurations need to be considered and the evaluation of each of these configurations takes a significant run-time.

We will now compare the computed buffer sizes, end-to-end latency and run-time for the two approaches using the WLAN 802.11p transceiver application. These numbers will be used to draw conclusions about which analysis method to use in which case. For each approach three different settings will be used as shown in Table 1: buffer sizing, introducing sequence constraints, and a combination of both. We also included results of an analysis option (#7) where no pruning was performed to identify the difference in run-time. Moreover, stopwatches are used in option #8, which can results in a speedup, but potentially also less accurate analysis results.

For the timed automata-based analysis method we list the number of configurations that need to be verified in the last column of Table 1. In case buffer sizing and introducing

Table 1: Settings used for the two analysis approaches and resulting configurations to consider.

#	Analysis method	Setting	Configurations or iterations
1	Approximative	Iterative buffer sizing	2
2	Approximative	Iter. sequence constr.	2
3	Approximative	Iterative combined	2
4	UPPAAL	Buffer sizing	768
5	UPPAAL	Sequence constraints	128
6	UPPAAL	Combination	8168
7	UPPAAL	Comb. no pruning	98304
8	UPPAAL	Comb. stopwatches	8168

Table 2: Analysis results obtained using the configurations in Table 1 for two minimization goals: minimizing buffer sizes and latency.

	Minimizing buffers		Minimizing latency		
#	$\sum$buffer	Latency (us)	$\sum$buffer	Latency (us)	Total run-time (s)
1	12	17	12	17	< 1
2	21	17	21	17	< 1
3	12	17	12	17	< 1
4	12	16	13	15	$2.2 \cdot 10^4$
5	21	15	21	15	$7.8 \cdot 10^4$
6	12	16	13	15	$9.7 \cdot 10^5$
7	12	16	13	15	$7.0 \cdot 10^6$
8	12	16	13	15	$4.8 \cdot 10^4$

sequence constraints are combined (#6) the number of configurations to consider is 12 times less than the product of both individual cases (98304, option #7). Firstly, the constraints that can be imposed by three of the seven sequence constraints becomes redundant when also performing buffer sizing and these sequence constraints are therefore removed. Secondly, after the task graph is translated into a dataflow graph, the minimum throughput can be verified. In this case, all configurations where the size of the buffer FFT to CHEST is less than two are discarded since the throughput is then limited by the sequentially executed tasks on the cycle FFT-EQ-DEMAP-DEINT-VIT-REENC-CHEST-FFT highlighted in red in Figure 9b, which cannot keep up with the periodic source. All remaining configurations (8168) could keep up with the source without the source blocking on full buffers. The approximative analysis approach did, for this example, always converge after two iterations as shown in Table 1.

The results of the analysis of the WLAN 802.11p transceiver are shown in Table 2. Two minimization goals: minimal buffer sizes and minimal latency, are presented in separate parts of the table. The approximative timed-dataflow approach, however, does not differentiate between these goals since its goal is to minimize buffer sizes.

The first part of the table shows that when minimizing buffer sizes, both the approximative timed-dataflow approach as UPPAAL-based approach are able to reach the optimum of 12, even without making use of additional sequence constraints. Although both approaches result in the same minimum total buffer size, the analysis using UPPAAL results in a lower latency.

The second part of the table shows that the minimum latency is only reached using UPPAAL, and is equal for all its analysis options (#4 to #8). Direct control over the interference between tasks using sequence constraints is advantageous in this case, since the least amount of configurations are considered (128 for option #5) and it results in the lowest run-time. The approximative timed-dataflow approach does result in a higher latency than our timed automata-approach, and interestingly reaches this latency of 17 μs for a different configuration where the buffer sized are smaller than the configuration as determined by UPPAAL with a corresponding latency of 15 μs. The difference between the best configuration for both approaches regarding latency is only in the buffer from FILTER to FFT, where the minimum latency is obtained for a buffer size of 2, where the approximative timed-dataflow approach used a buffer size of 1. We also analyzed the configuration of buffer sizes as found by the approximative timed-dataflow approach by using UPPAAL to compare the analysis results. This resulted in a latency of 16, which shows that higher buffer capacities, as derived by our approach, can lead to a lower latency. Also when using sequence constraints to optimize the latency, there is a difference between the two approaches. The approximative timed-dataflow approach sets the number of tokens on all forward edges to 0 and 1 for all backward edges. The minimum latency obtained by using UPPAAL is for a configuration where the backward edge from DEMAP to CHEST is different and contains 2 tokens instead of 1 for the approximative timed-dataflow approach. In case stopwatch automata are used (#7), the same minimal latency is found. However, latency analysis results are more pessimistic as a result of over-approximations for configurations where FFT can pre-empt EQ, i.e. the size of the buffer between FFT and EQ is larger than one.

The improvement in analysis results of our approach compared to the approximative timed-dataflow approach comes at the cost of a significant increase in run-time. The approximative timed-dataflow approach is always finished well within 1 second as shown in the last column in Table 2. On the other hand, the verification time in UPPAAL of each analysis option is between $2.2 \cdot 10^4$ and $9.7 \cdot 10^5$ seconds for option #4-#6 as shown in Table 1. The run-time is increased by one order of magnitude when the pruning step is not used as is shown for option #7 in Table 2. Using stopwatches (#8), a speedup of a factor 20 is achieved. When running 16 threads in parallel on a multicore server (2x Intel Xeon CPU E5-2630 v3 @ 2.40GHz), the total run-time for the option where sequence constraints are introduced (#4) is about 23 minutes.

9. CONCLUSION

In this paper we presented a hybrid analysis approach that determines the minimum latency of a cyclic task graph by adapting buffer sizes and sequence constraints using a combination of model checking and approximative analysis techniques. Each task is scheduled on one of the processors using a FPP scheduling policy.

Computationally efficient dataflow analysis techniques are used to derive lower and upper bounds on buffer sizes. In this way, the number of buffer size configurations is reduced that need to be considered by more accurate, but computationally intensive, model checking using UPPAAL. To be able to perform model checking, a network of timed automata is generated for each of the remaining configurations. The

latency of each configuration is determined using UPPAAL and the best configuration is selected.

Next to the cyclic dependencies resulting from blocking buffers, additional sequence constraints are inserted. Bounds on the number of initial tokens on these sequence constraints is determined by a slightly modified version of an approximative buffer sizing algorithm. These sequence constraints can potentially reduce the latency of a task graph.

We compared the results of our hybrid analysis approach with a state-of-the-art approximative dataflow analysis approach, which uses an iterative buffer sizing technique. Using our approach, the analyzed latency decreased from 17 μs to 15 μs. The decrease in latency is obtained at the cost of a run-time of 23 minutes instead of a fraction of a second.

10. REFERENCES

[1] 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 *Int'l Conf. on Formal Modeling and Analysis of Timed Systems*, pages 60–72. Springer, 2003.

[2] A. Brekling, M. R. Hansen, and J. Madsen. Models and formal verification of multiprocessor system-on-chips. *The Journal of Logic and Algebraic Programming*, 77(1-2):1–19, 2008.

[3] A. David, J. Illum, K. G. Larsen, and A. Skou. Model-based framework for schedulability analysis using UPPAAL 4.1. *Model-based design for embedded systems*, 1(1):93–119, 2009.

[4] R. de Groote, P. K. F. Hölzenspies, J. Kuper, and H. Broersma. Back to basics: Homogeneous representations of multi-rate synchronous dataflow graphs. In *Int'l Conf. on Formal Methods and Models for Codesign (MEMOCODE)*, pages 35–46. IEEE, 2013.

[5] J. Falk, J. Keinert, C. Haubelt, J. Teich, and S. Bhattacharyya. A generalized static data flow clustering algorithm for MPSoC scheduling of multimedia applications. In *ACM Int'l Conf. on Embedded Software (EMSOFT)*. ACM, 2008.

[6] E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Schedulability analysis of fixed-priority systems using timed automata. *Theoretical Computer Science*, 354(2):301–317, 2006.

[7] M. G. Harbour, J. G. García, J. P. Gutiérrez, and J. D. Moyano. MAST: Modeling and analysis suite for real time applications. In *Euromicro Conf. on Real-Time Systems (ECRTS)*, pages 125–134. IEEE, 2001.

[8] J. P. H. M. Hausmans and M. J. G. Bekooij. A refinement theory for timed-dataflow analysis with support for reordering. In *ACM Int'l Conf. on Embedded Software (EMSOFT)*, page 20. ACM, 2016.

[9] M. Hendriks and M. Verhoef. Timed automata based analysis of embedded system architectures. In *Proc. Parallel & Distributed Processing Symp.*, pages 8–pp. IEEE, 2006.

[10] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis-the SymTA/S approach. *IEE Proc. of Computers and Digital Techniques*, 152(2):148–166, 2005.

[11] B. Jonsson, S. Perathoner, L. Thiele, and W. Yi. Cyclic dependencies in modular performance analysis. In *ACM Int'l Conf. on Embedded Software (EMSOFT)*, pages 179–188. ACM, 2008.

[12] P. Krčál and W. Yi. Decidable and undecidable problems in schedulability analysis using timed automata. In *Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 236–250. Springer, 2004.

[13] G. Kuiper and M. J. G. Bekooij. Latency analysis of homogeneous synchronous dataflow graphs using timed automata. In *to appear in Design, Automation and Test in Europe (DATE)*. IEEE, 2017.

[14] G. Kuiper, S. J. Geuns, J. P. H. M. Hausmans, and M. J. G. Bekooij. Compositional temporal analysis method for fixed priority pre-emptive scheduled modal stream processing applications. In *Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pages 98–107. ACM, 2016.

[15] P. S. Kurtin, J. P. H. M. Hausmans, and M. J. G. Bekooij. Combining offsets with precedence constraints to improve temporal analysis of cyclic real-time streaming applications. In *Real-Time and Embedded Technology and Applications Symp. (RTAS)*, pages 1–12. IEEE, 2016.

[16] G. Madl, N. Dutt, and S. Abdelwahed. A conservative approximation method for the verification of preemptive scheduling using timed automata. In *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE*, pages 255–264. IEEE, 2009.

[17] M. Mikučionis, K. G. Larsen, J. I. Rasmussen, B. Nielsen, A. Skou, S. U. Palm, J. S. Pedersen, and P. Hougaard. Schedulability analysis using uppaal: Herschel-planck case study. In *Int'l Symp. on Leveraging Applications of Formal Methods, Verification and Validation*, pages 175–190. Springer, 2010.

[18] S. Perathoner, E. Wandeler, L. Thiele, A. Hamann, S. Schliecker, R. Henia, R. Racu, R. Ernst, and M. G. Harbour. Influence of different system abstractions on the performance analysis of distributed real-time systems. In *ACM Int'l Conf. on Embedded Software (EMSOFT)*, pages 193–202. ACM, 2007.

[19] R. Reiter. Scheduling parallel computations. *Journal of the ACM (JACM)*, 15(4):590–599, 1968.

[20] L. Thiele, S. Chakraborty, M. Gries, A. Maxiaguine, and J. Greutert. Embedded software in network processors-models and algorithms. In *ACM Int'l Conf. on Embedded Software (EMSOFT)*, pages 416–434. Springer, 2001.

[21] L. Thiele and N. Stoimenov. Modular performance analysis of cyclic dataflow graphs. In *ACM Int'l Conf. on Embedded Software (EMSOFT)*, pages 127–136. ACM, 2009.

[22] P. S. Wilmanns, S. J. Geuns, J. P. H. M. Hausmans, and M. J. G. Bekooij. Buffer sizing to reduce interference and increase throughput of real-time stream processing applications. In *IEEE Symp. on Real-Time Computing (ISORC)*, pages 9–18. IEEE, 2015.