A Recursive Approach to End-To-End Path Latency Computation in Heterogeneous Multiprocessor Systems

Simon Schliecker, Rolf Ernst
Institute of Computer and Communication Engineering
Technische Universität Braunschweig, Germany
[schliecker | ernst] @ ida.ing.tu-bs.de

ABSTRACT

This paper proposes a method for the derivation of end-to-end delays of applications that involve processing on multiple components in a heterogeneous multiprocessor system. The procedure precisely captures the pipelined and parallel processing of multiple events along an application path by accurately capturing the resource timing and avoiding the pay-bursts-only-once problem. Both time-triggered and event-triggered task activation schemes with arbitrary event patterns are supported.

In contrast to previous work, complex application topologies are allowed: The approach considers path forking and merging, as well as functional cycles and non-functional cyclic dependencies. The basis for the proposed method is an iterative compositional performance analysis, that allows computing event models in such systems. Based on the event models and local performance abstractions we propose a recursive approach to the derivation of the worst-case latency.

Categories and Subject Descriptors C.3 [**Special-Purpose and Application-based Systems**]: Real-time and embedded systems **General Terms**: Performance, Verification **Keywords**: real-time, multiprocessor, path latency

1. INTRODUCTION

Commonly, real-time systems that meet the computational requirements of today's demanding applications are composed from a variety of heterogeneous components, such as processors, busses, or hardware units. This composition optimizes important aspects of a design: The component specialization delivers high performance at a reduced cost and the application can be parallelized among the involved components which leads to an increased throughput. When such a system is expected to execute under real-time constraints, its temporal behavior needs to be investigated for compliance of the resulting timing with the application's requirements. For this, formal methods are without alternative when it comes to the reliability of their results.

CODES+ISSS'09, October 11–16, 2009, Grenoble, France.

To allow the application of such methods in a productive environment, several factors are important beyond the proven correctness: The results need to be as accurate as possible, because any overestimation will directly translate into an overdimensioning of the actual components, which ultimately means higher costs. Furthermore, to allow design space exploration, the analysis needs to be time-efficient and scale well to realistic system topologies and sizes.

Event-driven task activations have often been seen as a problem for the analysis of such multi-component systems, as local response time analysis can only be performed when the timing of the task activating events is known. In systems with cyclic dependencies, there is often no sequence of local analyses that allows a direct solution. For this reason, compositional analysis approaches [23, 4, 18, 8] rely on *event models* to express the possible event timing and derive conservative event models through iteration.

This paper proposes a two-step approach to system-level performance analysis. In a first step, the task activating event models are derived throughout the system. For this, the amount of dynamic, event-triggered task activations is bounded by event models that are provided by a compositional performance analysis. On this basis accurate system-wide information can be derived, namely the end-to-end path latency of applications that involve the processing of multiple tasks on multiple processors. The proposed approach greatly improves the accuracy of simpler approaches by acknowledging the effect of pipelined processing and the fact that overload situations are of transient nature. It avoids the pay-bursts-only-once problem that is present in other analyses. Still, we provide a versatile analysis, by allowing to combine a heterogeneous set of components and scheduling policies, with complex application topologies including functional cycles and non-functional cyclic dependencies.

The remainder of this paper is structured as follows: First, we present and evaluate the work related to the proposed approach in Section 2. We then introduce the new method to derive end-to-end latencies via multiple tasks on multiple processors that allows arbitrary event models and considers event pipelining in Section 3. We turn to diverging and merging path topologies in Section 4 and cyclic dependencies in Section 5. We conclude the paper with experiments (Section 6) and our conclusion in Section 7.

2. RELATED WORK

The performance analysis problem is addressed by various compositional approaches that separate the problem into

local component analyses and the modeling of event traffic between them. In Network Calculus [11] and the Real-Time Calculus [4] based on it, the local resource behavior is modeled as the execution time provided to the processing of events of a certain stream within a time window of given size Δt. Such a resource curve is depicted in Fig. 1a, for minimum ($\beta^-(\Delta t)$) and maximum supplied service ($\beta^+(\Delta t)$). The approach derives output event models and remaining resource capacity by folding operations in continuous time domain.

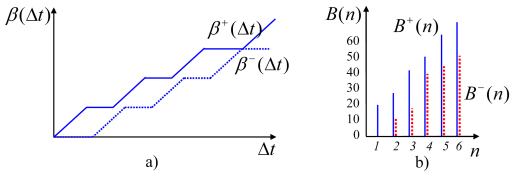

Figure 1: Models of Resource Service.

The opportunities of relying on simpler event and resource models have been explored in [6]. Here, the basic metric to model the real-time performance of the components are the tasks' worst (and best) case response times. This simple metric has been the focus of numerous research in single processor scheduling theory such as [10][1][24]. A common procedure for its derivation is symbolic simulation of a critical instant scenario. Various extensions have been proposed to improve the analysis results (e.g. by considering offsets [13] or variable task execution times [12]) and consider realistic scheduling behavior (e.g. FlexRay protocol [16], cache related preemption delay [20]).

A different approach to multiprocessor analysis is chosen in the holistic approaches of [25][5][15] where the classical single-processor scheduling theory is systematically extended and can be tailored toward a specific combination of input event model, resource sharing and communication policy. The global view on the system allows to take global correlations into account. However, in the case of a large number of such dependencies, the complexity of the analysis grows with system size and heterogeneity. In practice, deterministic networks such as TDMA are therefore highly useful to simplify the analysis procedure. Our analysis will not rely on such a holistic view, but rather perform a hierarchical analysis, extracting all relevant information from local resources before composing them on the system level.

The timing behavior of a system can also be modeled with the help of dataflow graphs in [3]. Properties such as the buffer requirements or the throughput of a system are then derived on the basis of max-plus algebra [2]. While the approach delivers accurate results for static systems and predictable arbitration policies, it does not allow to capture general schedulers in which the guaranteed service supplied to one application depends on the load imposed by another.

All of the above approaches bring a method to compute the end-to-end latency of events that are processed by sequential tasks on multiple resources. The simplest way to conservatively determine the latency is by accumulating the local worst-case response times as is done in e.g. [22] and [6]. However, this procedure is inaccurate in the case of bursty event occurrence owing to the 'pay-bursts-only-once" problem. A burst of events can in general occur at the input of any task along a path — leading to large local worst case response times — but the same event processed along the path can not experience this delay at each task.

Better estimates can be achieved through the convolution of component behavior along the path as is done in [11][4]. However, these methods rely on the concept of continuous time service curves. In general the folding operations can therefore be computationally intensive. The approaches have not been extended to cover analysis dependencies of functional cycles. The proposition of Section 3 is to perform similar folding operations in discrete time domain using the multiple event busy time. This naturally limits the computed values to the critical candidates.

In [7] an efficient method to compose the latency of pipeline stages is presented. The key idea is the derivation of a substitute single processor system which is then investigated for schedulability. However, it is specifically aimed at homogeneously scheduled systems, and does not allow for any cyclic functional or non-functional dependencies.

2.1 System Model

Tasks represent a sequence of operations with known minimum and maximum execution time. A task is activated by an *event*, and produces an event before the execution of the activation is finished. Tasks are mapped to resources that arbitrate between the tasks mapped to it according to their scheduling policy, which causes task activations to possibly interrupt each other.

2.2 Event Models

A key element of compositional performance analysis is the expression of the traffic flow between different components with the help of *event models*. In [4] and [6] event models describe the maximum and minimum number of events η that may occur during a time interval of given size Δt. Figure 2 shows such an event model representation on the left.

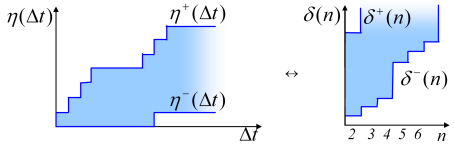

Figure 2: Event Model Representation.

An event model can also be expressed by the distances of the contained events. This is shown on the right side of Figure 2. The functions $\delta^-(n)$ and $\delta^+(n)$ represent the minimum and maximum distance between the occurrence of any n events in the stream. The δ functions are therefore meaningfully defined only for $n \geq 2$. Both the δ and the η representation can be converted to each other, such that the $\delta^-(n)$ function can be represented by the $\eta^+(\Delta t)$ function and vice-versa (the same is true for $\delta^+(n)$ and $\eta^-(\Delta t)$). The conversion can be done as follows:

$$\delta^-(n) = \inf_{0 \leq \Delta t, \Delta t \in \mathrm{R}} \{\Delta t \mid \eta^+(\Delta t) = n\} \quad (1)$$

$$\eta^+(\Delta t) = \max_{2 \leq n, n \in \mathrm{N}} [\{n \mid \delta(n) < \Delta t\} \cup \{1\}] \quad (2)$$

In this paper we will mainly utilize the δ representations. This representation has the strong advantage of being a discrete function of n, rather than the η representations, which

are continuous. This allows us to conveniently investigate a discrete set of relevant events in our formulas, rather than operating with continuous functions, in which only the steps contain true information.

Correct δ functions have some fundamental properties (e.g. super-additivity [11]), but no rules are imposed on the parameters with which they are actually represented. For a compact description, the *standard event models* in [17] rely on the three parameters event stream period $\mathcal{P}$, event stream jitter $\mathcal{J}$, and minimum distance between any two events d^{min}. The δ-function of e.g. a bursty event stream can then be expressed as follows:

$$n \geq 2 : \delta^+(n) = (n-1)\mathcal{P} + \mathcal{J} \quad (3)$$

$$\delta^-(n) = \max((n-1)d^{min}, (n-1)\mathcal{P} - \mathcal{J}) \quad (4)$$

2.3 Resource Model

In [1, 24], the busy period is defined as the time interval from the occurrence of a "critical instant" until the resource is idle for the first time. One of the task activations within this busy period then experiences the worst case response time R^{max}. This concept has been generalized in [19], where the *multiple event busy time* function was introduced to represent the amount of time necessary to process a certain number of events that arrive within the same busy window. For example, $B^+(1)$ is the maximum busy window inflicted by a single event that arrives after the previous was finished. $B^+(2)$ is the maximum busy window size that is spanned by two events, where the second arrives before the first is finished. The minimum busy time B^- is defined correspondingly.

Definition 1 (Multiple Event Busy Time)**.** *The n-event busy time $B_i^+(n)$ ($B_i^-(n)$) of a task i is given by the maximum (minimum) time it may take i to process n events, given that all but the first of the n events arrived before the preceding was finished.*

Figure 1b) shows an example minimum and maximum busy time function. Comparable to the *service curves* in network calculus, the multiple event busy time is *independent* of the actual activating event model of the investigated task i.

As an example, Theorem 1 provides the multiple event busy time for static priority preemptive scheduling of independent tasks without preemption costs. However, the multiple event busy time can also be provided for other scheduling policies [19].

Theorem 1. *The multiple event busy time $B_i^+(n)$ for a task i under static priority preemptive scheduling with independent tasks is given by*

$$B_i^+(n) = n \cdot C_i + \sum_{j \in hp(i)} (\eta_j^+(B_i^+(n)) \cdot C_j$$

where

C_i, C_j is the maximum core execution time of task i, j.

$hp(i)$ is the set of tasks with higher priority than i.

$\eta_i^+(\Delta t)$ is the maximum number of events that lead to an activation of task i in a time window of size Δt.

A proof for this theorem can be found in [19]. Equation 5 can be solved iteratively until a fixed-point has been found.

2.4 Compositional Performance Analysis

In the multiprocessor performance analysis of [6] the analysis of individual components is interleaved with the propagation of event models. This procedure is repeated until a fixed-point representing conservative estimates of the event traffic anywhere in the system is found. This framework is shown in Figure 3. A description of the analysis procedure follows.

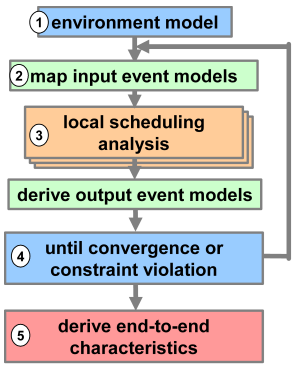

Figure 3: Compositional Analysis Loop.

First, the environmental input event models representing the minimum and maximum amount of events that the system is exposed to are specified (1). All other input event models within the system are initialized with optimistic starting points, which are iteratively refined during the analysis procedure. These event models are supplied to the individual resources (2), where they are used for local analysis (3). This local analysis provides timing information for each task mapped to the resource (such as the best-case and worst-case response times (R^{min} and R^{max}) based on single processor scheduling theory [10][24], or the more expressive multiple event busy time $B^+(n)$ and $B^-(n)$ introduced in the previous section).

Based on the components timing information, the output event models are calculated. These output event models can in turn be input event models to other components, or outputs to the environment. The output event models are compared to those used in the previous analysis iteration (4). If all are the same the analysis has converged, otherwise the corresponding local analyses are repeated with the refined inputs.

All event models can only become more generic with each iteration [18, 21, 9], meaning that each iteration contains the previous models. Thus, the complete procedure is monotonic. The analysis is complete if either all event streams converge toward a fixed-point, or if an abort condition, e.g. the violation of a timing constraint has been reached.

In this paper we assume that all task activating event models are known. Thus, in the case of event-driven task activations and dynamic scheduling, a compositional performance analysis has preceded the investigation of the path latencies as described in the following sections.

3. PATH LATENCY

A common problem in real-time systems is that multiple tasks on the same or different processors are subsequently involved in the processing of an event. For example, multiple controllers can be involved in a sensor-actor chain.

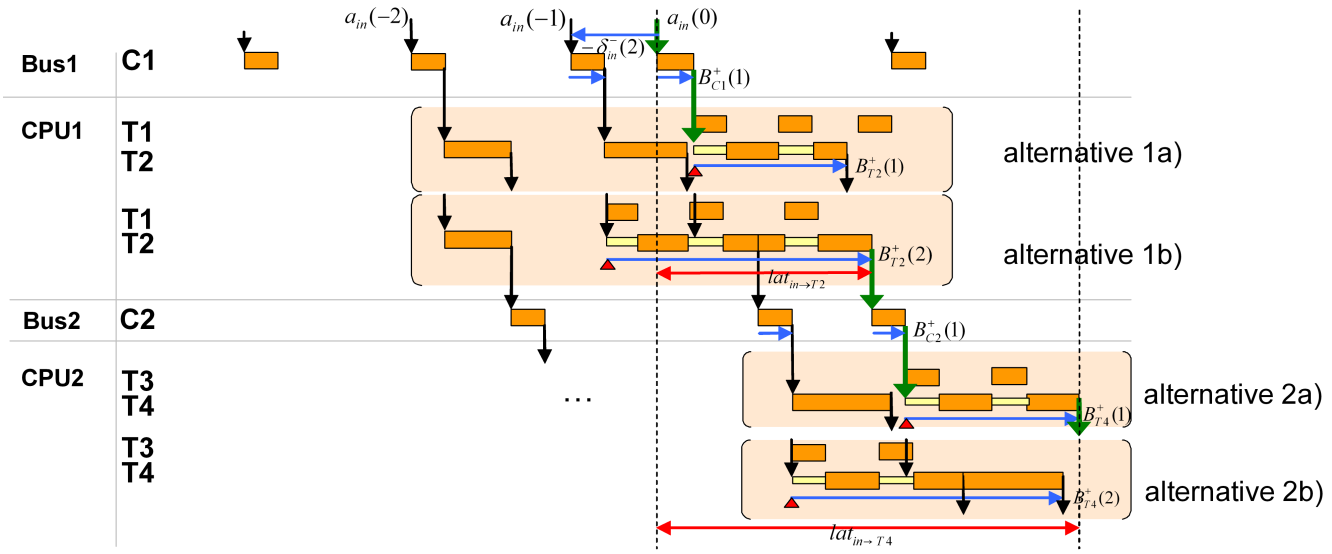

Figure 4: Example For Worst-Case Path Latency of Event 0 along the path $\{C0, T1, C2, T4\}$.

But also streaming applications have throughput constraints that are determined by the sequential processing on multiple resources. Such a processing chain opens opportunities to use specialized components, and benefit from increased throughput through event pipelining. However, these benefits can only be exploited for real-time systems, if an accurate analysis is available that captures the timing behavior.

The classical approach to derive the end-to-end latency [6][22] has been to accumulate the individual task worst-case response times along the path. This simple summation is then a conservative estimate of the end-to-end latency. But obviously it also leads to a large overestimation in the case of bursty event arrivals: If a burst enters the system this translates into large local worst-case response times, as an event may have to wait for preceding events of the same stream to be finished. Usually, such a burst can occur anywhere along the considered path, and consequently all local worst-case response times will be relatively large. In reality however, an event that has been delayed by its predecessors on one resource can not be fully delayed by the predecessors again on the successive resource. During its waiting time the preceding events have continued to be processed on the successive resources.

Note that the calculated local worst-case response times and traffic estimates may be correct and conservative, only can they not be experienced by the same event traversing a path. This effect has been called the "pay-burst-only-once" phenomenon [11]. Similar to the approach in network calculus, we avoid this problem by providing a dedicated analysis for the complete path that considers the correlation between token arrival times and local response times. We build on this analysis to consider more complex topologies such as merging and diverging paths and functional cycles. The rationale behind the improved path latency analysis is explained in the following example.

3.1 Example

Consider a system with 2 CPUs and 2 Busses, for which a possible gantt diagram is depicted in Figure 4. A critical message needs to be periodically transported via the path $\{C0, T1, C2, T4\}$, and arrives already with a small jitter. On the two CPUs, higher priority tasks $T2$ and $T3$ are periodically activated, with no known correlation to $T1$ and $T4$ (such as offsets).

Once the messages have been transported over $Bus1$, two scenarios are possible that may lead to a worst-case latency of an arbitrary event 0:

1a) The interference by the higher priority task $T1$ is aligned with the arrival of event 0 (indicated by the small triangle), and thus the corresponding activation will experience the worst-case multiple event busy time $B^+_{T2}(1)$.

1b) The interference by $T1$ is aligned with the arrival of the preceding event -1, and thus the corresponding activation is delayed by the unfinished previous activation. In this case both events -1 and 0 have been processed $B^+_{T2}(2)$ after the arrival of event -1.

Task activations further in the past may not interfere in this example due to a sufficient distance between the activating events. Scenario 1b) produces a later production time of event 0 at the output of $CPU1$:

$$lat_{in \to T2} = \max[0 + B^+_{C1}(1) + B^+_{T2}(1), \\ 0 - \delta_{in}(2) + B^+_{C1}(1) + B^+_{T2}(2)] \quad (5)$$

This reasoning can be continued for the subsequent resources. On $Bus2$, the additional latency of event 0 is straight-forwardly bounded by $B^+_{C2}(1)$. There can be no interference from preceding activations, and thus no scenarios need to be checked.

On the next component, $CPU2$, there are again two relevant scenarios which may cause the largest latency increment for event 0 to be processed by task $T4$. Event 0's finishing time is maximized, if the interference of task $T3$ is aligned with the arrival of event 0, as depicted in Scenario 2a. Aligning the interference with the arrival of the previous event, as shown in Scenario 2b, can not lead to a larger production time of event 0 at the output of $T4$.

The worst-case path latency for event 0 is in this example given by Equation 6, in which the different possible scenarios are listed. Only two events may interfere either on $CPU1$ or $CPU2$, because the arrival of three events is always further apart than the time to process two events on any resource.

$$lat_{in \to T3} = \max[\quad (6)$$
$$0 + B^+_{C1}(1) + B^+_{T2}(1) + B^+_{C2}(1) + B^+_{T4}(1),$$
$$0 - \delta_{in}(2) + B^+_{C1}(1) + B^+_{T2}(2) + B^+_{C2}(1) + B^+_{T4}(1),$$
$$0 - \delta_{in}(2) + B^+_{C1}(1) + B^+_{T2}(1) + B^+_{C2}(1) + B^+_{T4}(2)]$$

The concept of the worst-case path latency can be generalized to the worst-case processing time for a number of events. The classical latency is then a special case of this n-event latency for $n = 1$. This metric can be useful, for example, when a series of samples needs to be collected before a valid output can be calculated. Also, it allows to consider multiple events in larger time frames, which allows to contain the influence of transient effects.

3.2 Pipelined Path Latency

First, we will introduce the concept of *causal dependence*, which provides the relationship between events at the input of a task and those produced by it. For the scope of this paper we assume that each task is *activated* once at the moment when one event has arrived at each of its input ports ("AND activation semantic"), it is then *ready* until it has been assigned sufficient time on the processor and then *terminates*. Each task activation produces exactly one event at each of its output ports before it terminates.

All events are numbered according to the sequence of their occurrence — events occurring later receive higher numbers. We will later focus on an arbitrary event 0, thus preceding events will have *negative* numbers. The arrival time of an event n at the resource to which task i is mapped is denoted with $e_{i-1}(n)$. The time at which the resulting task activation produces an event is denoted by $e_i(n)$. Tasks process the events of an event stream in-order. This is a typical assumption in scheduling theory matching the design practice. Prioritized events are modeled with separate event streams.

Along a path of tasks, defined in the following definition, the output events of one task become the input events that activate the successive task.

Definition 2 (Path). *A* path, $\mathrm{P} = \{T_1, T_{end}\}$ *is a fully ordered set of tasks between the path beginning* T_1 *and the path end* T_{end}*, such that for each task* $i \in \mathrm{P}$*, each activation of* i *is causally dependent on an activation of its predecessor* $i - 1$*.*

Definition 3 (Causal Dependence of Events). *An event* b *is* causally dependent *on event* a*, if* b *is produced by the same task activation that consumes* a*, or the task activation that produces* b *is causally dependent on the task activation that consumes* a*.*

Definition 4 (Causal Dependence of Task Activations). *A task activation* B *is* causally dependent *on another task activation* A*, if* B *is activated by an event that is causally dependent on the event that has led to* A*.*

Definition 5 (Numbering). *A task activation is activated for the* n*-th time when it has received* n *events at each of its inputs. Each task activation produces exactly one event with the same number.*

With these definitions, we can formally define the path latency as illustrated in the above example.

Definition 6 (Path Latency). *The* n*-event end-to-end latency of path* $\mathrm{P} = \{T_1, T_{end}\}$*, is defined as the maximum distance between the arrival of an arbitrary event* 0 *at the input of* T_1 *and the production of the* n*-th causally dependent event at the output of the last task* T_{end}*.*

$$lat_{\mathrm{P}}(n) = \quad \max \quad e_{end}(n-1) - e_0(0) \quad (7)$$

The maximum value of $lat_{\mathrm{P}}(n)$ depends on the actual timing of the events that are processed along the path as well as the timing of all other events in the system. In our approach, we abstract the timing of the other events with the help of the multiple event busy time model per involved task, which represents the local worst-case behavior. We compose these local results to derive a maximum value for $lat_{\mathrm{P}}(n)$ that takes into account the inherent pipelining of event processing along the path.

Because the events arriving at the first task of the path belong to the same event stream, their arrival times are correlated, so that their minimum (and maximum) distances are given by their comprising event model. Thus, if the arrival time of one event is known, all preceding and successive events have a certain minimum and maximum distance from this event according to (8) and (9).

$$a_i(p) \geq a_i(q) + \delta_i^-(q - p + 1) \quad \forall p, q \in \mathrm{N}, p < q \quad (8)$$
$$a_i(p) \leq a_i(q) + \delta_i^+(q - p + 1) \quad \forall p, q \in \mathrm{N}, p < q \quad (9)$$

These inequations can be used to bound the latest possible arrival times of events at any task where the input event model is known. For the derivation of path latencies, this bound is particularly interesting at the beginning of a path (i.e. for $i = 1$).

Lemma 1. *With respect to the arrival of an event* 0 *at task* i*, the event* n*, with* $n \neq 0$ *arrives at task* i *no later than*

$$e_{i-1}(n) \quad \leq \quad \begin{cases} e_{i-1}(0) - \delta^-(-n+1), & \text{if } n < 0 \\ e_{i-1}(0) + \delta^+(n+1), & \text{if } n > 0 \end{cases}$$

where $\delta^-(n)$ *(*$\delta^+(n)$*) is the minimum (maximum) distance between any* n *events arriving at task* i*.*

Proof. All events leading to activations of task i are constrained by their corresponding event stream, which defines a minimum $\delta^-(k)$ and maximum $\delta^+(k)$ distance between the occurrence of any k events. Any event preceding event 0 ($n < 0$) has a minimum distance to the occurrence of event 0, and can thus not arrive later than $e_{i-1}(0) - \delta^-(-n+1)$. Any event succeeding event 0 ($n > 0$) has a maximum distance to the occurrence of event 0, and can thus not arrive later than $e_{i-1}(0) + \delta^+(n+1)$. □

The following lemma provides an upper bound on the time at which an arbitrary activation of a task i in the system is finished ($e_i(n)$), relative to its activation ($e_{i-1}(n)$), and the preceding activations of the same task ($e_{i-1}(n-k)$).

Lemma 2. *The exit time* $e_i(n)$ *of any event* n *produced by task* i *is bounded by*

$$e_i(n) \leq \max_{k \geq 0} \{e_{i-1}(n-k) + B_i^+(k+1)\} \quad (10)$$

where

- $e_{i-1}(n-k)$ *is the arrival time of the k-th event before event n.*
- $B_i^+(k+1)$ *is the maximum multiple event busy time for* $k+1$ *events to be processed by task i.*

PROOF. The output event n is produced at time $e_i(n)$ by the activation of task i that is activated by the event n that arrives at time $e_{i-1}(n)$. There are two cases: Either the previous activation of task i is finished when the input event arrives, or it is not.
Case 1: $e_{i-1}(n) > e_i(n-1)$ (previous activation finished). In this case, following the definition of the busy time, the activation n is finished no later than $e_{i-1}(n) + B_i(1)$.
Case 2: $e_{i-1}(n) < e_i(n-1)$ (previous activation not finished). The input event n arrives while at least one previous event has not been produced. Let k be the number of events that have not been processed. Thus, $e_i(n-k-1) < e_{i-1}(n) \leq e_i(n-k)$. (We assume that an event arriving at the very instant at which the previous activation terminates does not fall into the same busy interval.) In this case, the multiple event busy time bounds the time at which the busy interval that was started by event $n-k$ (and to which the event n now contributes to) is over: $e_i(n) < e_{i-1}(n-k) + B(k)$. k is unknown in general and depends on the runtime behavior, but it is conservative to assume the maximum exit time over all possible values of k. □

The numerical computation of (10) requires the comparison of a possibly infinite number of event arrival times and resulting busy times. However, the set of possible k's is well bounded, because during the worst-case response time R_i^{max} of i only a limited number of activations may occur (formally, an upper bound for k is given by $k < \delta_{i-1}^-(R_i^{max})$). Another bound was proposed in [19], which is based on the fact that the maximum busy time function B_i^+ grows sub-additively, while the minimum distance between events at the input δ_{i-1}^- grows super-additively.

Using Lemma 1 and Lemma 2, it is possible to derive the absolute values for the occurrence of events along a path with respect to the arrival of event 0 at the input of the beginning of the path. This allows computing the path latency as defined in Definition 6 in the following theorem.

Theorem 2. *The worst-case latency of path* $\mathrm{P} = \{T_1, T_{end}\}$ *can be recursively calculated as follows:*

$$lat(n) = e_{end}(n-1) - e_0(0) \quad (11)$$

$$e_i(n) \leq \max_{k \geq 0}\{e_{i-1}(n-k) + B_i^+(k+1)\} \quad (12)$$

$$e_0(n) \leq \begin{cases} e_0(0) - \delta^-(-n+1), & \text{if } n < 0 \\ e_0(0) + \delta^+(n+1), & \text{if } n > 0 \end{cases} \quad (13)$$

PROOF. The theorem follows directly from Definition 6, Lemma 2, and Lemma 1. □

Theorem 2 allows to compute the worst-case path latency for $n = 1$ events, as well as the maximum amount of time between the arrival of event 0 at the path beginning, and the production of the n-th causally dependent event at the path end.

The minimum end-to-end latency can be calculated with a similar method. However, for most realistic cases the minimum latency is simply the sum of the best case response times — it is therefore not explored further in this paper.

4. FORK AND JOIN APPLICATION TOPOLOGIES

Typical applications in embedded systems consist of more than just a sequence of tasks that is sequentially activated on the arrival of new data. In an automotive application for example, data may be needed from different sensors before computation can begin, and the computed results may be distributed to multiple actuators afterwards. In multimedia, intermediate parallelizations are very common, for example, when the frame of a video stream is split into a number of subframes that are then processed in parallel on an array of hardware elements, before the result is again merged for further processing. Such applications commonly feature *join* and *fork* structures.

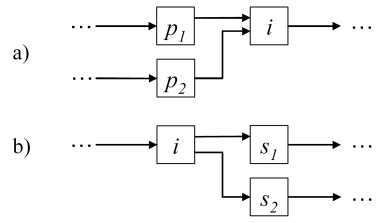

Figure 5: Possible application topologies: a) join structure , b) fork structure

To address these application structures, the computation of the event exit times from Lemma 2 can be extended. We have defined that a task a) is activated for the n-th time no later then when n events have arrived at each of its inputs and b) produces the n-the event on each of its outputs no later than when its n-th activation is finished. Firstly, a task with multiple outputs will produce one event on each output when it has finished execution.

Lemma 3 (Fork)**.** *Let a task i have multiple outputs and let* S_i *be the set of successor tasks that are connected to these outputs via edges* $i \rightarrow s$*. If i finishes the activation n at time* $e_i(n)$ *then event n arrives at the connected input of each of the successor tasks no later than*

$$\forall s \in S : e_{i \rightarrow s}(n) = e_i(n) \quad (14)$$

PROOF. Follows directly from the definition of the task behavior. □

When a task has multiple inputs in the given model it is activated when it has received sufficient events on each of its inputs. The following lemma bounds the event production of a task i that has multiple inputs. For this an intermediate variable $e'_{i-1}(n)$ is introduced, which represents the time at which task i has received sufficient events from its predecessors to begin its n-th activation.

Lemma 4 (Join)**.** *Let task i have multiple inputs, and* P_i *be the set of the direct predecessor tasks of i at each of its inputs, then the event n is produced by task i no later than*

$$e_i(n) \leq \max_{k \geq 0}\{e'_{i-1}(n) + B_i^+(k+1)\} \quad (15)$$

with

$$e'_{i-1}(n) \leq \max_{p \in P_i}[e_p(n)] \quad (16)$$

PROOF. The task will be ready to execute its n-th activation when it has received an equal number of n events from each of its direct predecessors. This will be the case no later than $e'_{i-1}(n)$. Once an activation is ready, it will be processed after $B_i^+(k+1)$, depending on the amount k of unfinished preceding activations. The reasoning now follows the proof of Lemma 2. □

The events arriving at the different inputs of a task may directly or indirectly come from different sources that may or may not be related to each other. If some inputs of i have a common root in a task r, the same production times $e_r(n)$ of events at the common root are used to compute the event arrival times $e_p(n)$ from all subpaths. The maximum operation in (16) will then automatically consider the longest subpath.

Some of the inputs of i may also have independent roots, i.e. when data from different sensors is required for an activation, but that data is sent without synchronization. The calculation of the worst-case event arrival times will then require the token arrival times of tasks that have no further predecessors. One of these tasks without predecessors will be T_1, i.e. the beginning of the investigated path P = $\{T_1, T_{end}\}$, for which $e_0(n)$ is defined in Theorem 2. Other tasks without predecessors, denoted as "external sources", may be completely unrelated. Each of these sources must essentially have the same long-term throughput to avoid buffer overflow at one of i's input. Still, the different sources may produce data with different patterns or jitter, which causes an activation delay and increases the latency along path P. The magnitude of this delay is given by the distance between the arrival of an event along path P and the arrival of the corresponding event on any of the other inputs. Let P_i^{ext} be the set of direct predecessors of a task i that are not causally dependent on a task in path P. Then the maximum delay to the activation of task i due to such external events is bounded by the possible "drift" between the arrival of events:

$$D_i \leq \max_{n\geq 2} \max_{p\in P_i^{ext}} [\delta_p^+(n) - \delta_{i-1}^-(n)] \qquad (17)$$

where $\delta_p^+(n)$ is the maximum distance between events arriving at the predecessor p, which has been computed during the compositional performance analysis step as explained in Section 2.4. With this, we can rephrase (16) independently of path external inputs:

$$e'_{i-1}(n) \leq \max_{p\in P_i \setminus P_i^{ext}} [e_p(n)] + D_i \qquad (18)$$

Computing (17) can be computationally elaborate, depending on the parameters of the underlying event model. In [8] the delay D_i was analytically derived for the case where the event models are represented as standard event models.

5. CYCLIC DEPENDENCIES

In a general multiprocessor setup, functional cycles and non-functional cyclic dependencies may exist between different tasks. These disrupt a straight-forward analysis procedure as proposed in the previous section. However, our goal is to admit both types of cyclic dependencies in order to increase the scope of systems that can be investigated. Our solution is two-fold: First, non-functional dependencies are resolved by deriving the respective task activating event models and compute the task's multiple event busy times. Secondly, functional cycles are tackled through a stop condition that breaks an infinite recursion.

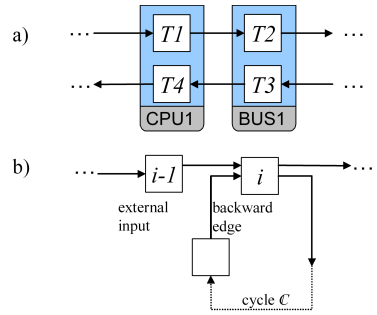

Figure 6: Cyclic Dependencies: a) non-functional cyclic dependency, b) functional cycle

5.1 Non-functional Cyclic Dependencies

Non-functional dependencies are the implicit result of resource sharing in a system. Figure 6a) for example shows a system with two tasks mapped to different processors. Assume a priority driven scheduling and let $T4$ and $T2$ have the highest priorities on their respective resources. Then the response time of $T1$ can not be computed without knowledge of $T3$'s output event model, and $T3$'s output event model can not be computed without an analysis of $T1$. Such cyclic dependencies are common in larger setups with multiple processors and buses.

To tackle these dependencies, iterative analysis procedures have been proposed that derive the actual event models in the system through a fixed-point approach (see Section 2.4, and [18, 4, 9, 21]). After the analysis has converged, task activating event models δ_i^-, δ_i^+ are known for every task in the system, whether it is time-triggered or event driven. These event models are then the basis for computing for each task the worst-case response time R_i^{max} and multiple event busy time function B_i^+. With the help of these functions, the path latency can be computed according to Theorem 2.

5.2 Functional Cycles

Functional dependencies are inherent to the application structure, and independent of the hardware mapping. A typical example of a functional cyclic dependency is a control application that consists of a loop that contains a sampling operation (*sensor*), the computation of a new parameter (*controller*), and the controlling actuators (*plant*). Here, sufficient throughput is required for control stability. Loops are also common in signal processing applications, where the result of a preceding activation is reused as an input for the current filtering step.

In systems with such functional cycles, the recursive application of Lemma 2 or Lemma 4 will cause an infinite recursion, because an activation of a task i in a cycle depends on the finishing time of a preceding activation of i. This problem can be addressed by bounding the necessary number of recursions. The main idea is that in a *schedulable* system, the processing of an event in the cycle can not be delayed by events that have arrived sufficiently far in the past.

Formally, a cycle exists if a task activation n is causally

dependent on a preceding activation of the same task. Let task i have an "external input" at which events arrive at times $a_i(n)$, an output edge that goes to the successive task along a cycle C, and a second input edge that comes from the preceding task in C. Given that a path P is investigated, of which task i is a part, then the external input will be the one that comes from its predecessor $i-1$ along the path ($a_i(n) = e_{i-1}(n)$). Task i then requires events on both the external input and the backward edge in order to be activated. If this happens we say that the event n is *admitted* into the cycle.

The dependency constraint imposed by the limited resources in a cycle can be modeled with *tokens* which limit the number of external events that have been admitted into the cycle but for which the corresponding events (*tokens*) have not returned to the backward edge. For a given cycle C, there are a number of tokens which either correspond to events generated by preceding activations, or represent initial placement of data that is required to admit the first several events. We know from [2] that the number of tokens in a cycle remains constant during execution, because each activation of a task on the cycle will consume exactly one token on its input and produce one token on its output edge. Let d_C be the number of such tokens for cycle C, then the n-th activation of a task $i \in \mathrm{C}$ is causally dependent on the $n - d_\mathrm{C}$-th activation of i.

Lemma 4 provides an upper bound on the time that the n-th activation of task i is finished, which is when i has received events on all its inputs, including the backward edges of the cycles C of which it is a part. Based on this lemma, we define that an event n is *admitted* into the cycle at time $a'_i(n)$.

$$a'_i(n) \leq \max[a_i(n), \max_{\mathrm{C} \in C} e_\mathrm{C}(n - d_\mathrm{C})] \tag{19}$$

$e_\mathrm{C}(n-d_\mathrm{C})$ is a function of the time that task i was started for the $n - d_\mathrm{C}$-th time and the time that was necessary to process token $(n - d_\mathrm{C})$ by tasks on the cycle. In order to bound the latter, we first establish an upper bound on the amount of time a token may be processed in a cycle.

Lemma 5 (Response Time Bound in Cycles)**.** *In a cycle* C *with a total number of d_C tokens, the response time of an arbitrary task $j \in \mathrm{C}$ is bounded by $B_j(d_\mathrm{C})$.*

Proof. Each activation of j consumes one token from its input edge and produces one on its output edge sometime later before it terminates. Thus the number of tokens on the cycle s constant d_C, and there may be no more than d_C simultaneous activations of task j at any time. Furthermore, none of these activation is influenced by successive activations due to in-order processing of the activations. Therefore, the maximum time for any activation to finish is bounded by the multiple event busy time $B_j(d_\mathrm{C})$. □

Obviously, if a task j is part of multiple cycles, its response time can not be larger than $B_j(d^{min})$, where d^{min} is the smallest d_C of all cycles $\mathrm{C} \in C$ that j is a part of. Lemma 5 can directly be used to bound the round-trip time of tokens in a cycle, i.e. the latest possible time after the admission of an event n into the cycle that it has taken the corresponding token to be returned to the backward edge.

Lemma 6 (Round-Trip in Cycles)**.** *In a cycle* C *with a total number of d_c tokens, any token has traversed the cycle no later than $\sum_{j \in \mathrm{C}} B_j(d_\mathrm{C})$:*

$$e_\mathrm{C}(n) \quad \leq \quad a'_i(n) + \sum_{j \in \mathrm{C}} B_j(d_\mathrm{C}) \tag{20}$$

Proof. Follows directly from Lemma 5. □

To derive the time $a'_i(n)$ at which event n is admitted into the cycle, we need to check whether it is influenced by preceding events that have not been processed by the cycle. With the help of Lemma 6, we can establish that the processing of previous tokens on a cycle C does not impose a constraint on the n-th activation of task i, when

$$a_i(n) \geq a'_i(n - d_\mathrm{C}) + \sum_{j \in \mathrm{C}} B_j(d_\mathrm{C}) \tag{21}$$

In order to find out whether condition (21) is indeed fulfilled for an event n, we need to check further into the past, whether the admission of event $(n - d_\mathrm{C})$ into the cycle was itself delayed by preceding tokens, which again may be delayed by their predecessors $(n - qd_\mathrm{C})$. The constraint to break this recursion now relies on the fact that the minimum distance to the arrival of previous events grows superadditively, while the possible delay that these past tokens may impose only grows linearly. Eventually thus, the distance to the arrival of previous events will be sufficiently large to rule out any interference with the present activation.

This concept is formalized in the following theorem which bounds the number of successive events that may not be immediately admitted into a cycle.

Lemma 7 (Admittance of Preceding Events)**.** *Let a task i be part of a cycle* C *and have an external input at which events arrive at times $a_i(n)$ with a minimum distance of $\delta_i^-(m)$ between any m arriving events. Then:*
If there is a $k \geq 1$ that fulfills the following inequation

$$\delta_i^-(kd_\mathrm{C} + 1) \quad > \quad k \sum_{j \in \mathrm{C}} B_j(d_\mathrm{C}) \tag{22}$$

there must be for any event n at least one preceding event j with $j = n - qd_c, 1 \leq q \leq k$ that was immediately admitted:

$$\exists j, j = n - qd_c, 1 \leq q \leq k : a'_i(j) = a_i(j) \tag{23}$$

Proof. The proof is by contradiction. Assume that all preceding events $n - qd_\mathrm{C}, 1 \leq q \leq k$ arrive at the external input in such a way that the corresponding tokens on the cycle have not been returned to the backward edge, and thus no incoming event is immediately admitted:

$$\forall q, 1 \leq q \leq k : a_i(n - qd_\mathrm{C}) < e_\mathrm{C}(n - (q+1)d_\mathrm{C}) \tag{24}$$

Let the arbitrary time between the arrival time and the admittance time of the first of the events considered in the theorem be denoted with D and thus:

$$a'_i(n - kd_\mathrm{C}) \quad = \quad a_i(n - kd_\mathrm{C}) + D \tag{25}$$

By iterative application of (19) and (20) under the assumption (24), we can deduce

$$a'_i(n) \quad \leq \quad a'_i(n - kd_\mathrm{C}) + k \sum_{j \in \mathrm{C}} B_j(d_\mathrm{C}) \tag{26}$$

and with (25)

$$a'_i(n) \quad \leq \quad a_i(n - kd_\mathrm{C}) + k \sum_{j \in \mathrm{C}} B_j(d_\mathrm{C}) + D_n \tag{27}$$

Also, we know from the properties of the incoming event stream that events arrive at the input with a certain minimum distance according to (8), and thus

$$a_i(n) \geq a_i(n - kd_C) + \delta_i^-(kd_C + 1) \quad (28)$$

and because of $a_i(n) \leq a'_i(n), \forall n$:

$$a'_i(n) \geq a_i(n - kd_C) + \delta_i^-(kd_C + 1) \quad (29)$$

Equations (27) and (29) imply

$$\delta_i^-(kd_C + 1) \leq k \sum_{j \in C} B_j(d_C) + D_n \quad (30)$$

In order to fulfill the condition of the theorem (equation (22)), D_n would need to be negative, which is impossible because of (25) and $a_i(n) \leq a'_i(n), \forall n$. Thus, we have to conclude that the assumption (24) is not fulfilled for at least one q and the theorem follows. □

Theorem 7 can now be used to break the infinite recursion in the analysis of the admittance time $a'_i(n)$ of an event n by assuming

$$a'_i(n - kd_C) = a_i(n - kd_C) \quad (31)$$

where k is the smallest integer that fulfills condition 22. From this, the admittance times of the successive events follow. If no such k exists, the cycle can not be deemed schedulable, because the distance between incoming events could then be steadily smaller than what can be processed by the involved tasks in the cycle.

The smallest k that fulfils (22) can be computed numerically for any event model, or based on the parameters of the specific event model. For example, in the standard event model of equations (3) and (4) with parameters period P and jitter J, k can be upper bounded by the following closed-form inequation:

$$k \leq \frac{Pd_c - J}{\sum_{j \in C} B_j(d_C) - Pd_C} \quad (32)$$

With the computable bound on the activation and finishing times of any task in the system, we have provided a versatile method to compute the maximum end-to-end latency — considering topological hurdles such as fork and join constructs, non-functional dependencies, and functional cycles.

6. EXPERIMENTS

We have conducted a set of experiments to show the validity and precision of the presented approach. Consider the path $\{S0, S1\}$ in the example of Figure 7 in which events are processed along a diverging path via 6 tasks on 2 buses and 3 CPUs. Each task may be disturbed by higher priority events from another application. The higher priority application is activated by source $S2$ and requires data from a previous iteration to be available. External events arrive at $S2$ with an average distance (period) of 100 and a jitter of 200, and at $S0$ with a period of 20.

The results of two parameter scenarios are plotted in Figure 8: In one scenario, all execution times are constantly 5 ("Constant ET"), in the other scenario all execution times along path $\{S0, S1\}$ are variable between 1 and 5 ("Variable ET"). In either case, the response times are not constant, because of the dynamic interference from the higher priority application.

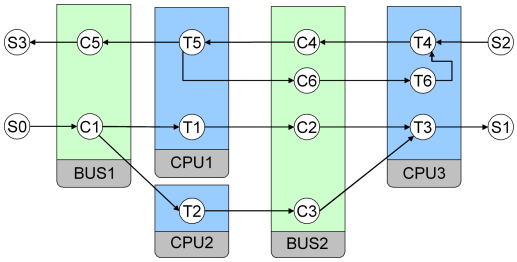

Figure 7: Example System.

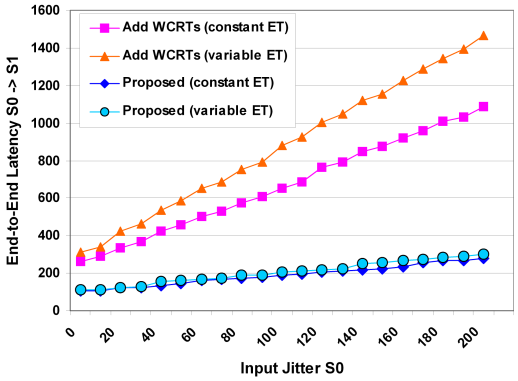

Figure 8: Comparison of Path Latencies

The latency calculation that is based on the accumulation of local worst cases ("Add WCRTs") draws the expected overestimation from the pay-bursts-only-once problem. The effect becomes more substantial with growing timing uncertainty (i.e. increasing input jitter at source S0, or introduction of variable execution times). The proposed analysis ("Proposed") tackles this problem through correlating the local worst case busy times. It is also relatively insensitive to the increase of dynamism. In the given parameter range the proposed method calculates a between 59% and 79% better end-to-end latencies.

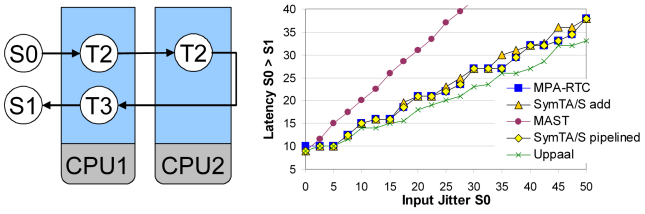

Figure 9: Example System and Comparison of End-to-End Latency (from [14]).

Further experiments of small systems have been performed in the scope of the comparative paper [14], in which real-time calculus [4], SymTA/S [18], and other approaches were compared. The path latency computation of this paper was included in the experiments. In one experiment a short chain of 3 tasks on 2 CPUs is investigated. A particular challenge to the analysis is the correlation of events activating T3 and those activating T1. It can be seen in Figure 9 that the new path latency calculation "SymTA pipelined" is better than the simple additive calculation (SymTA add"). Of all

approaches under comparison, our approach is in this experiment closest to the actual worst cases derived with model checking ("Uppaal"), often matching it accurately. This was owed to the fact that the analysis on CPU1 better considers the offsets between the activations of T2 and T3 according to e.g. [13]. [14] also contains an evaluation of analysis times — besides its accuracy, our analysis is also very fast.

The number of operations that will be performed to derive the path latency depends on the number of tasks in the path and the size of the busy time interval of each task, i.e. the number k of possibly coinciding events in equation 12 due to their input event model. In the worst case, all distributions of the coincidences on the local resources may be checked (e.g. in the example of Figure 4 with 2 relevant tasks and a maximum coincidence of 2 events, 3 busy time combinations are possible). In general, given a maximum number of b interfering events along a path of length l this leads to a maximum number of $\binom{(l-1)+b}{b}$ operations. In practice the number is smaller, because the slower tasks dominate over many candidates. In the given example that contains alternative paths, the total number of comparisons was between 31 in the case of no input jitter for $S0$ and 65 in the case where the jitter was 200 (which corresponds to 10 coinciding events at the input).

7. CONCLUSION

This paper has presented an efficient methodology to derive end-to-end path latencies in a multiprocessor system with heterogeneous components and complex application topologies. The method is suitable to consider arbitrary event models in large variety of different heterogeneous scheduling policies. The approach considers pipelining and transient overload effects that surface along the path.

In experiments, we have demonstrated that the derived end-to-end latencies are quantitatively on par with specialized approaches based on network calculus. In addition, the proposed recursive analysis process supports a larger spectrum of application topologies including functional cycles and non-functional dependencies.

8. REFERENCES

[1] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[2] F. Baccelli, G. Cohen, G. Olsder, and J. Quadrat. *Synchronization and linearity*. Wiley New York, 1992.

[3] M. Bekooij, O. Moreira, P. Poplavko, B. Mesman, M. Pastrnak, and J. van Meerbergen. Predictable embedded multiprocessor system design. *Proceeding of the SCOPES workshop, September*, 2004.

[4] S. Chakraborty, S. Kunzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. *Proc. 6th Design, Automation and Test in Europe (DATE)*, pages 190–195, 2003.

[5] J. Gutiérrez, J. García, and M. Harbour. On the Schedulability Analysis for Distributed Hard Real-Time Systems. *Proceedings of the 9th Euromicro Workshop on Real-Time Systems, Toledo, Spain*, pages 136–143, 1997.

[6] 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.

[7] P. Jayachandran and T. Abdelzaher. A Delay Composition Theorem for Real-Time Pipelines. *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 29–38, 2007.

[8] M. Jersak, K. Richter, and R. Ernst. Performance analysis for complex embedded applications. *International Journal of Embedded Systems*, 1(1):33–49, 2005.

[9] B. Jonsson, S. Perathoner, L. Thiele, and W. Yi. Cyclic dependencies in modular performance analysis. In *Proceedings of the 8th ACM international conference on Embedded software (EMSOFT)*, pages 179–188, New York, NY, USA, October 2008. ACM.

[10] M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390, 1986.

[11] J. Le Boudec and P. Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer, 2001.

[12] A. Mok and D. Chen. A multiframe model for real-time tasks. *IEEE Transactions on Software Engineering*, 23(10):635–645, 1997.

[13] J. Palencia and M. Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proc. 19th IEEE Real-Time Systems Symposium (RTSS98)*, 1998.

[14] S. Perathoner, E. Wandeler, L. Thiele, A. Hamann, S. Schliecker, R. Henia, R. Racu, R. Ernst, and M. G. Harbour. Influence of different abstractions on the performance analysis of distributed hard real-time systems. *Design Automation for Embedded Systems*, pages 1–23, April 2008.

[15] P. Pop, P. Eles, and Z. Peng. Schedulability analysis and optimization for the synthesis of multi-cluster distributed embedded systems. *Design Automation and Test in Europe Conference and Exhibition (DATE)*, pages 184–189, 2003.

[16] T. Pop, P. Pop, P. Eles, Z. Peng, and A. Andrei. Timing Analysis of the FlexRay Communication Protocol. *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 203–213, 2006.

[17] K. Richter and R. Ernst. Event Model Interfaces for Heterogeneous System Analysis. In *Proc. Design Automation and Test in Europe (DATE)*, 2002.

[18] K. Richter, D. Ziegenbein, M. Jersak, and R. Ernst. Model composition for scheduling analysis in platform design. In *Proceedings of the 39th conference on Design automation (DAC)*, pages 287–292. ACM New York, NY, USA, 2002.

[19] S. Schliecker, J. Rox, M. Ivers, and R. Ernst. Providing accurate event models for the analysis of heterogeneous multiprocessor systems. In *Proc. Intl. Conference on Hardware/Software codesign and system synthesis (CODES+ISSS)*, pages 185–190. ACM New York, NY, USA, 2008.

[20] J. Staschulat, S. Schliecker, and R. Ernst. Scheduling Analysis of Real-Time Systems with Precise Modeling of Cache Related Preemption Delay. *Real-Time Systems, 2005.(ECRTS 2005). Proceedings. 17th Euromicro Conference on*, pages 41–48, 2005.

[21] S. Stein, J. Diemer, M. Ivers, S. Schliecker, and R. Ernst. On the convergence of the symta/s analysis. Technical report, Technische Universität Braunschweig, Braunschweig, Germany, November 2008.

[22] J. Sun and J. Liu. Bounding the end-to-end response time in multiprocessor real-time systems. *Proceedings of the 3rd Workshop on Parallel and Distributed Real-Time Systems (WPDRTS)*, 1995.

[23] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. *Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on*, 4, 2000.

[24] K. Tindell, A. Burns, and A. Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Systems*, 6(2):133–151, 1994.

[25] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 40(2-3):117–134, 1994.