Response-Time Analysis for Task Chains in Communicating Threads

Johannes Schlatow and Rolf Ernst
Institute of Computer and Network Engineering, TU Braunschweig
{schlatow,ernst}@ida.ing.tu-bs.de

***Abstract*—When modelling software components for timing analysis, we typically encounter functional chains of tasks that lead to precedence relations. As these task chains represent a functionally-dependent sequence of operations, in real-time systems, there is usually a requirement for their end-to-end latency. When mapped to software components, functional chains often result in communicating threads. Since threads are scheduled rather than tasks, specific task chain properties arise that can be exploited for response-time analysis. As a core contribution, this paper presents an extension of the busy-window analysis suitable for such task chains in static-priority preemptive systems. We evaluated the extended busy-window analysis in a compositional performance analysis using synthetic test cases and a realistic automotive use case showing far tighter response-time bounds than current approaches.**

I. Introduction

Large embedded systems are often implemented as a collection of functions, each described as a task graph. To derive end-to-end response times in such task graphs, we are typically interested in the response times between the respective tasks. The problem of task graph schedulability analysis and deadline feasibility has been intensively studied, most recently in [1], where a task-graph classification was provided in the context of system types, along with the respective schedulability analysis algorithms and their complexity. In this paper, we are interested in task chains which are derived from communicating software threads leading to specific task chain properties that can be exploited for response-time analysis covering both synchronous and asynchronous communication. Such communicating threads have become the common implementation vehicle, e.g. in automotive software components [2] or in microkernel-based systems [3], [4].

A further challenge is the inclusion of this analysis in a global system-level timing analysis that covers larger systems integrating tasks with different scheduling policies. Therefore, the response-time analysis should be compatible to a compositional performance analysis, such as *Real-Time Calculus (RTC)* [5] or *Compositional Performance Analysis (CPA)* [6]. Unfortunately, in such an analysis, even simple task chains currently lead to quite conservative results (cf. Section VIII), such that new chain analysis solutions are required.

As a core contribution, this paper presents an extension of the busy-window response-time analysis which is able to cope with varying priorities along the chain resulting from thread communication. We will see that chaining tasks with arbitrary priorities incurs priority-inversion problems which lead to deferred load challenging the busy-window mechanism. In the end, we will apply the results to a synthetic example and to a realistic automotive system with communicating threads.

II. Modelling Communicating Threads

In communicating software threads, we can identify two different communication or activation semantics: the **synchronous IPC** and the **asynchronous notifications**. The former is the conventional communication type in microkernel-based systems and resembles a procedure call where the caller blocks and waits for the reply of the callee. The latter relates to a message-based (buffered) communication much like a "fire & forget" scheme in which the sender transmits a notification to the receiver without being blocked in its own execution. As a result, asynchronous notifications can queue up at the receivers's input.

While the asynchronous scenario is already common in timing analysis, the nature of the synchronous scenario is – to our knowledge – currently not well-reflected in common timing analysis models. Furthermore, when we try to perform a timing analysis for these systems, we first encounter a mismatch between the programming model and the timing analysis model: On the one hand, the programmer implements a thread that communicates at arbitrary points in its execution using the available primitives, whereas the timing architect models the system by tasks that are only allowed to communicate at the end of their execution, implicitly assuming asynchronous communication semantics.

Figure 1 and 2 illustrate a possible model transformation for the scenarios of synchronously and asynchronously communicating threads. According to [7], this transformation is supposed to close the gap between the two models. Figure 1 shows a thread (Thread 1) that (synchronously) calls another thread (Thread 2) at some point in its execution. Thread 1 can only continue after the latter completed and returned. The right side of the figure illustrates how this scenario is reflected in the timing model by a sequence of tasks that represent the segments of the threads ($1a$, 2 and $1b$) along with their precedence relations. Figure 2 depicts a similar scenario but with an asynchronous notification instead of the call. Here, the second segment of Thread 1 does not need to wait for the completion of Thread 2. In the resulting timing model the task τ_{1a} thus activates both tasks (τ_2 and τ_{1b}) simultaneously.

Although this is a straightforward transformation, it already obfuscates important information that should be incorporated

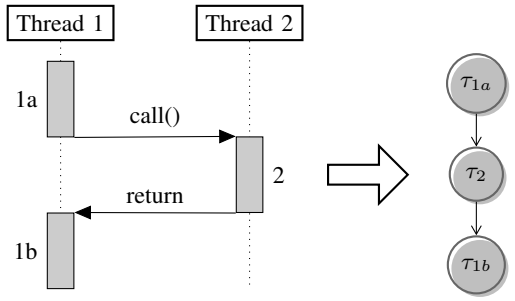


Figure 1. Communicating threads (implementation) naturally split up into a chain of tasks (timing model).

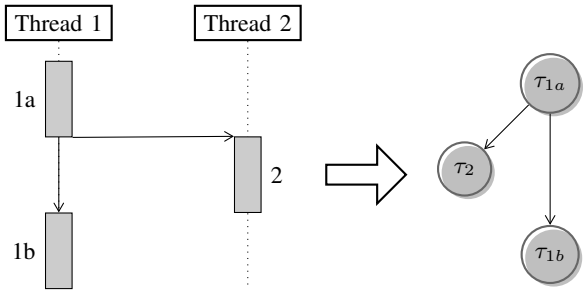


Figure 2. Transformation of an asynchronous notifications between threads into the timing model.

in the timing analysis: As already mentioned, the task timing model does not reflect the blocking behaviour of the synchronous call, i.e. it does not reflect that τ_{1a} cannot execute again before τ_2 returned. In addition, it neither represents the execution dependencies between the thread segments, i.e. that τ_{1a} cannot execute before τ_{1b} finished.

One way to approach this with conventional timing analysis – assuming a static priority scheduling – is to assume an ascending priority assignment for a task chain such that a task is never interfered by its predecessor. However, this strongly contradicts what can actually be implemented on a real system where a thread (e.g. Thread 1) is actually the scheduled entity and thus determines the priority of several tasks in the timing model (e.g. τ_{1a} and τ_{1b}). In consequence, it is not possible to assume a monotonic priority assignment for a task chain that models communicating threads but rather enforces an alternating pattern between higher and lower priorities. However, as we demonstrate in Section VIII, we encounter (very) pessimistic results if we perform timing analysis on such realistic priority assignments.

Instead, we show that by augmenting the timing analysis model with the activation semantics we can derive a proper analysis approach for these scenarios that not only improves the latency estimates but also requires much less computational effort.

This paper therefore comprises the following **contributions**:

- An extended timing analysis model that covers the semantics of synchronous and asynchronous activations reducing the number of event-model propagations in a Compositional Performance Analysis (CPA).
- A worst-case response time analysis for entire task chains in static-priority preemptive systems based on the busy-window approach.
- Improved worst-case end-to-end latency bounds for synchronous as well as asynchronous chains of tasks.

As we base our approach on the CPA, we first summarise its essentials in Section III before elaborating on the related work in Section IV. In Section V, we introduce our extension to the timing analysis model and provide the required assumptions and definitions. We then present and refine our response-time analysis for synchronous task chains in Section VI and further show its application to asynchronous chains as well (Section VII). Finally, we evaluate and compare the presented analyses in Section VIII before drawing our conclusion in Section IX.

III. Compositional Performance Analysis

In CPA [6], systems are modelled by (processing) resources, scheduling policies, tasks and precedence relations. The tasks are mapped to the **resources** on which they execute and consume service (processing time). As, typically, multiple tasks share the service provided by a resource, the resource's **scheduling policy** determines how the contention is resolved. A **task**'s execution behaviour is modelled by an activation, core execution and propagation step. Once activated, a task τ_i can be scheduled for its core execution on the resource according to the scheduling policy (e.g. static-priority preemptive). The **core execution** is modelled by the worst-case and best-case execution time, denoted C_i^+ and C_i^-, which provide the upper and lower bound on the workload induced by a single activation of task τ_i. Subsequent to its core execution, a task activates its dependent task(s) in the propagation step. Activations are propagated according to a directed graph of tasks (nodes) and **precedence relations** (edges) representing functional data dependencies such as communication primitives. In the CPA task model, activations are buffered and can queue up, i.e. the tasks can be executed independently from each other. As the actual data is of no interest, the timing relations between task activations are modelled by an **event model interface**, which abstracts every possible trace of events (i.e. activations) by its lower and upper bound. An event model is expressed by a pair of arrival curves $\eta^+(\Delta t)/\eta^-(\Delta t)$ defining an upper/lower bound on the number of events that can arrive within any half-open time window $[t, t+\Delta t)$ [8]. Alternatively, an event model can be represented by the pseudo-inverse functions, the so-called minimum/maximum distance function $\delta^-(n)/\delta^+(n)$, that return a lower/upper bound on the time interval between the first and the last event of any sequence of n event arrivals [9]. For the sake of brevity, we denote a *pair* of arrival curves/distance functions by a bold symbol and by omitting the superscript, i.e. $\boldsymbol{\eta}(\Delta t)/\boldsymbol{\delta}(n)$ respectively. Note that event models are also used to describe activations from external sources, such as a (periodic) timer or other devices.

In CPA, the system model and the environmental model define the task graph and the event models for external activations respectively. Initially, optimistic input event models

for all tasks are derived based on the environmental model. The actual analysis is then composed of iteratively performing **local resource analysis** and **event model propagation**, which propagates the newly calculated output event models in order to refine the input event models of dependent tasks. The analysis terminates once convergence or non-schedulability is reached.

The local resource analysis particularly yields the **worst-case response time** (WCRT) of all tasks mapped to the resource and serves as a basis for deriving the output event models. In the scope of this paper, we base the local resource analysis on the generalised busy-window technique as presented in [10] for preemptive static-priority scheduling as follows:

Definition 1. *(q-event busy-window) The q-event busy-window $B_i(q)$ denotes the maximum time a resource may be busy processing q events of task τ_i and is iteratively calculated by the following formula:*

$$B_i(q) = q \cdot C_i^+ + \sum_{j \in \mathcal{I}_i} \eta_j^+(B_i(q)) \cdot C_j^+ \quad (1)$$

where $\mathcal{I}_i$ denotes the set of interfering higher-priority (or equal-priority) tasks w.r.t. p_i.

Equation 1 assumes that all q events (except the first) arrive prior to the completion of the preceding events, i.e. before the resource becomes idle. Hence the maximum q for which this assumption holds is given by Equation 2 as follows:

$$Q_i = \max\{n : \forall q \in \mathbb{N}^+, q \leq n : \delta_i^-(q) \leq B_i(q-1)\} \quad (2)$$

Based on this, the WCRT of task τ_i is found among all $q \in [1, Q_i]$ busy windows as follows:

$$R_i^+ = \max_{q \in [1,Q_i]} (B_i(q) - \delta_i^-(q)) \quad (3)$$

In the remainder of this paper, we assume that the output event models are accurately derived from the q-event busy-window as described in [10].

After convergence of the analysis loop, the worst-case end-to-end latencies can be conservatively estimated by adding up the resulting WCRTs R_i^+ of all tasks τ_i belonging to a path of interest [6], [11].

IV. Related work

There are various compositional approaches that address the performance analysis problem by separating it into local component (resource) analyses and modelling of the communication behaviour between these components. One prominent approach is the *Real-Time Calculus (RTC)* [5], which models this behaviour by arrival curves and service curves in continuous time domain and applies convolutions in order to derive the system-level performance metrics (e.g. end-to-end latencies).

In contrast, CPA [6] relies on discrete time models to derive worst-case (and best-case) response times of every task on a resource (cf. Section III). The busy-window technique applied for this has been introduced in [12], [13] and generalised for arbitrary event models in [10].

Several improvements have been made in the past regarding the coverage of data dependencies (i.e. precedence relations) and their implied correlations. In [14] a recursive path analysis approach was presented which addresses the issue that a burst of input events shall only be "paid" once when estimating the latency for a particular path in a multiprocessor system. This approach still relies on a WCRT computation on the local resources but uses a recursive algorithm that improves the estimated latency by excluding impossible combinations of event arrivals and busy times within the path. On the other hand, *offset analysis* [15] is an approach that focuses on the improvement on the response-time analysis. Here, correlations resulting from a *transactional task model* are taken into account by introducing time offsets between different event arrivals. These offsets can be static or dynamic [16]. Although offset analysis is – in general – computationally intensive, efficient implementations have been achieved for periodic event models (with jitter) [17]. In [18] and [19] offset analysis has been leveraged to take precedence relation into account.

In contrast to the compositional approaches, one can also pursue a *holistic approach* [20] in order to take global correlations in a multiprocessor system into account. However, the complexity of such an holistic analysis grows with the size of a system and the number of contained dependencies. In the scope of this work, we thus still focus on a local rather than a holistic view on a real-time system.

Many of these analyses have been implemented by research-oriented [9], [21]–[23] as well as commercial [24] tools. An evaluation and comparison between the different abstractions for performance analysis can be found in [25].

In addition to the advances in performance analysis, [7] presented how formal timing analysis and verification can be integrated into the development process of real-time software by means of a model transformation. As this closes the semantic gap between the design and timing model, it augments the practicability of timing analysis in (industrial) design processes at least from the modelling perspective. Our work rests upon this model transformation but focuses on the quality and usability of the results (i.e. estimates) achievable in this context.

V. Extended timing analysis model

When we perform timing analysis – e.g. CPA as introduced in Section III – for task chains, we typically encounter some pessimism in the resulting end-to-end latencies. More precisely, the latencies are conservative but overestimate the worst-case. On the one hand, this overestimation originates from the event-model propagation which masks the timing correlations between activations of dependent tasks. At the expense of additional computational effort, offset analysis can be used in order to account for the timing correlations inherent in the precedence relations [18], [19]. On the other hand, the summation of the tasks' WCRTs to compute the end-to-end latency may further add the same source of interference multiple times (i.e. for each task) while, in reality, the same event processed by a chain of tasks cannot experience the same

interference at each task. This has also been addressed as the "pay-bursts-only-once" problem in [14].

When we look at the scenario of communicating threads in Figure 1, we note that there are strict constraints for the precedence and order of task executions. I.e. a synchronously activated task can only interfere once with its predecessor, as the latter must wait for the completion of its successor before being activated again. This fact contrasts the common task model, which assumes that activations can arbitrarily queue up on the edges (buffers) in the task graph.

We therefore need a sound timing analysis that respects the activation semantics we commonly encounter on our target systems. Here, "sound" denotes the following aspects:

- conservative
- little overestimation
- bounded computational effort

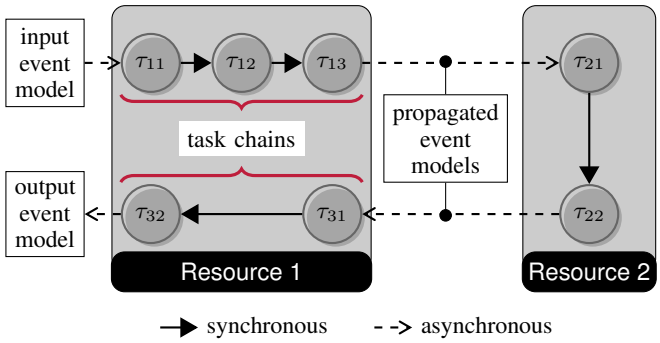


Figure 3. Extended task graph example

Based on this, we propose the following extension to the common timing analysis model. In this extended model, we differentiate between the activation semantics, i.e. we distinguish synchronous (i.e. blocking) from asynchronous (i.e. non-blocking) edges in the task graph. As our response-time analysis approach (Section VI) considers entire task chains as opposed to single tasks, we introduce a preprocessing step in which the task chains are defined. Furthermore, as we only need the propagated event models at the inputs/outputs of the task chains, this preprocessing basically dissects the task graph into task chains and the interjacent edges (e.g. resource boundaries), which we call *propagation points*. Figure 3 illustrates such an extended task graph. Note that the preprocessing can either be done manually by the designer/timing architect, or automatically by tool support. In the scope of this paper, we focus on a manual process that holds the following assumptions:

1) Task chains do not cross resource boundaries.
2) Task chains cannot fork, i.e. a task within a chain must not have multiple outgoing edges.
3) Task chains cannot join, i.e. a task within a chain must not have multiple incoming edges (future work).

Definition 2. *(Task chain)*

A task chain denotes a path in the task graph in which every node is mapped to the same resource and has an indegree and outdegree of 1. A synchronous *task chain only contains synchronous edges whereas an* asynchronous *task chain only contains asynchronous edges. A task chain might contain only a single task.*

Note that synchronous task chains reflect the scenario of communication threads while asynchronous task chains represent conventional precedence relations between tasks. Although we restrict task chains to a single activation type in the scope of this paper, the reasoning presented in Section VI and VII can also be applied to task chains with mixed activation semantics. Note that our analysis approach is not limited to CPA as it only modifies (extends) the local resource analysis. In a multiprocessor system it should furthermore be combined with an improved subsequent path analysis such as [14].

VI. Response-time analysis for synchronous task chains

In this section, we present the improved response-time analysis for synchronous task chains. Figure 4 depicts two synchronous task chains a and b and illustrates the notation we use for the remainder of this section. Every task τ_{ij} is specified by two indices. The first index (i) denotes the task chain it belongs to while the second index (j) declares the position of the task within the chain assuming a sequential numbering starting from zero. A task chain i (more specifically: task τ_{i0}) is further activated by its input event model $\boldsymbol{\eta}_i(\Delta t)$. The (propagated) output event model of a task chain i is denoted by $\tilde{\boldsymbol{\eta}}_i(\Delta t)$. Furthermore, let p_{ij} denote the priority of τ_{ij}.

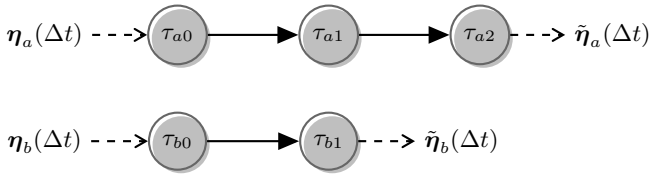


Figure 4. Two synchronous task chains a and b

In contrast to the classical CPA, our analysis approach pursues the idea of computing the WCRT of entire task chains in order to prevent that the same interference effect is pessimistically accounted for every task. In the context of the compositional analysis flow, this is a valid approach due to the fact that (propagated) output event models are only required between task chains as we proposed in Section V.

In order to derive the *task-chain busy window*, which computes the busy window of an entire task chain a as opposed to a single task (cf. Eq. 1), we first differentiate between three categories of interference:

- intra-chain interference ($\tau_{ij} \forall i = a$)
- inter-chain interference ($\tau_{ij} \forall i \neq a$)

For a synchronous task chain, we know that it cannot interfere with itself hence there is no intra-chain interference. Regarding the inter-chain interference, we can provide better bounds based on the following observations; note that we are still assuming a preemptive static-priority scheduling:

Assume a priority assignment for the task chains depicted in Figure 4 that satisfies $p_{a0} \geq p_{a2} > p_{b0} \geq p_{b1} > p_{a1}$. I.e. the tasks of chain b have a higher priority than the second task of chain a but a lower priority than the first and last task. In this case, τ_{a1}'s execution is blocked by task chain b potentially leading to a deferred activation of τ_{a2} as illustrated by the Gantt chart in Figure 5. Here, τ_{b0} is executed first, as there is no higher-priority activation pending, immediately followed by τ_{b1}. The latter, however, is preempted by τ_{a0} as soon as it is activated. Yet, task chain a cannot complete because τ_{a1} is blocked by task chain b, i.e. τ_{a1} can only execute after τ_{b1} completed (and if there is no pending activation of task chain b). The completion of τ_{a1} eventually releases τ_{a2}. When we now take a look at a second activation of task chain b, we observe that it may experience interference from the first and second activation of task chain a, more precisely from τ_{a2} as well as τ_{a0}. However, this interference can occur at most once as τ_{a1} is blocked by task chain b and therefore cannot release τ_{a2}. On the other hand, if we direct our focus on task chain a, we observe that this chain may be interfered arbitrarily by τ_{b0} and τ_{b1} due to the fact that they have a higher priority than τ_{a1}, which is a classical priority inversion effect.

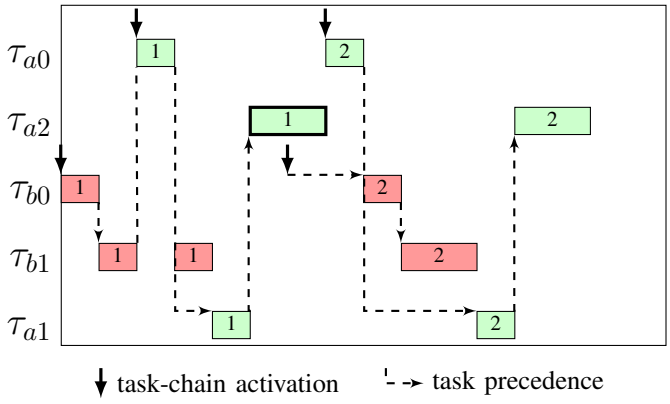


Figure 5. Gantt chart for a deferred synchronous activation of τ_{a2} (bold).

Note that priority inheritance [26] can also be implemented for synchronous task chains [27] in order to avoid the priority inversion but here we are treating the general case where this is usually not possible. By pursuing the generalised analysis approach for synchronous task chains, we can easily use it in order to similarly improve the analysis of asynchronous task chains as we present in Section VII. In the remainder of this section, we first introduce a simple but general bound for the (synchronous) task-chain busy window before we refine this further for the synchronous case.

A. Task-chain busy window

We now formulate the *task-chain busy window* for a task chain i similar to Eq. 1 taking the above observations into account. I.e. we split the inter-chain interference into tasks that can arbitrarily interfere and those for which we can guarantee that they can execute only once within the busy window of task chain i. First, we conservatively define the set of higher-priority tasks for each interfering task chain j by considering every task τ_{jk} that has a higher-priority than the lowest priority task in chain i. Note that we use a superscript c to denote symbols in the context of task chains in general whereas the superscripts sc and ac indicate a particular validity only for synchronous or asynchronous chains.

Definition 3. *$\mathcal{H}^c_{ij}$ denotes the set of higher-priority tasks (i.e. their indices) from task chain j w.r.t. task chain i.*

$$\forall j \neq i: \quad \mathcal{H}^c_{ij} = \{k | p_{jk} \geq \min_l p_{il}\} \tag{4}$$

If any of the tasks from task chain j is blocked, all higher-priority tasks are deferred and thus interfere at most once.

Definition 4. *(Deferred tasks)*
$\mathcal{D}^{sc}_{ij}$ denotes the set of tasks (i.e. their indices) in a synchronous task chain j that are deferred by the execution of task chain i, i.e. all tasks τ_{jk} that can only interfere once with task chain i.

$$\forall j \neq i: \quad \mathcal{D}^{sc}_{ij} = \begin{cases} \mathcal{H}^c_{ij} & \exists \tau_{jk} : k \notin \mathcal{H}^c_{ij} \\ \emptyset & \textit{otherwise} \end{cases} \tag{5}$$

Definition 5. *$\mathcal{I}^c_{ij}$ contains the indices of all higher-priority tasks in task chain j that are not deferred by task chain i.*

$$\forall j \neq i: \quad \mathcal{I}^c_{ij} = \mathcal{H}^c_{ij} \setminus \mathcal{D}^c_{ij} \tag{6}$$

with $\mathcal{D}^c_{ij} = \mathcal{D}^{sc}_{ij}$.

In our example (Figure 4 and 5), we thus get: $\mathcal{I}^c_{ab} = \mathcal{H}^c_{ab} = \{0, 1\}$, $\mathcal{D}^{sc}_{ab} = \emptyset$, $\mathcal{D}^{sc}_{ba} = \mathcal{H}^c_{ba} = \{0, 2\}$ and $\mathcal{I}^c_{ba} = \emptyset$.

It is worth emphasising, that we can reuse Eq. 6 for asynchronous or mixed task chains just by adapting $\mathcal{D}^c_{ij}$ appropriately.

Based on these definitions, we can come up with a simple bound for the (synchronous) task-chain busy window.

Corollary 1. *The q-event busy window for the synchronous task chain i is calculated by:*

$$B^{sc}_i(q) = q \sum_k C^+_{ik} + \sum_{j \neq i} \left(\sum_{k \in \mathcal{I}^c_{ij}} (\eta^+_j(B^{sc}_i(q)) C^+_{jk}) + \sum_{k \in \mathcal{D}^{sc}_{ij}} C^+_{jk} \right) \tag{7}$$

Similar to Eq. 1, the first sum term adds up the core execution time for q activations of the entire task chain. The second term eventually computes the inter-chain interference, which is split into the set of arbitrarily interfering tasks $\mathcal{I}^c_{ij}$, which are accounted $\eta^+_j(B^{sc}_i(q))$ times, and the "one-time" interferers $\mathcal{D}^{sc}_{ij}$.

B. Refined task-chain busy window

If we now consistently pursue the fact that any task that is blocked by task chain i cannot release its dependent tasks, we infer that a task chain j is broken up into several segments as

soon as there are multiple blocked tasks. We formally define these segments as follows:

Definition 6. *(Circular segment)*
A circular segment $\mathcal{S}_{ikl}$ is the set of tasks (their indices) between τ_{ik} and τ_{il} assuming that the last task is followed by the first task of the chain:

$$\mathcal{S}_{ikl} := \{m | (0 \leq m < n_i) \wedge ((k \leq m \leq l) \vee (m \geq k \geq l) \vee (k \geq l \geq m))\} \quad (8)$$

where n_i denotes the length of task chain i.

Moreover, we are only interested in those segments that exclusively comprise deferred tasks:

Definition 7. *(Deferred segments)*
The set of circular segments in task chain j that can interfere at most once with chain i is given by:

$$\mathcal{S}^*_{ij} := \{\mathcal{S}_{jkl} | 0 \leq k < n_j, 0 \leq l < n_j, \forall m \in \mathcal{S}_{jkl} : m \in \mathcal{D}^c_{ij}\}$$

with $\mathcal{D}^c_{ij} = \mathcal{D}^{sc}_{ij}$.

Theorem 1. *A task chain i can only be interfered by a single deferred segment $\mathcal{S}_{jkl} \in \mathcal{S}^*_{ij}$ of chain j.*

Proof. The proof is by contradiction, hence we assume that task chain i is interfered by two deferred segments, $\mathcal{S}_{j23}$ and $\mathcal{S}_{j56}$. By definition, τ_{j1} and τ_{j4} cannot execute in chain i's busy window. Furthermore, if $\mathcal{S}_{j23}$ interferes with task chain i, task chain j must have been preempted right after the execution of τ_{j1}. On the other hand, if $\mathcal{S}_{j56}$ interferes with chain i, chain j must have been preempted right after the execution of τ_{j4}. However, as τ_{j4} requires $\mathcal{S}_{j23}$ and τ_{j1} requires $\mathcal{S}_{j56}$ to execute beforehand, the hypothesis must be rejected. □

For a worst-case analysis, we are therefore interested in the *critical deferred segment*, which maximises the interference on chain i.

Definition 8. *(Critical deferred segment)*
The critical segment of task chain j w.r.t. chain i is the longest interfering segment in terms of execution time.

$$\forall j \neq i : \quad \mathcal{S}^{crit}_{ij} := \underset{\mathcal{S}_{jkl} \in \mathcal{S}^*_{ij}}{\arg\max} \left(\sum_{m \in \mathcal{S}_{jkl}} C^+_{jm} \right) \quad (9)$$

Corollary 2. *The q-event busy window for a synchronous task chain i only needs to consider the critical deferred segments instead of all deferred tasks and is calculated by:*

$$B^{sc}_i(q) = q \sum_k C^+_{ik} + \sum_{j \neq i} \left(\sum_{k \in \mathcal{I}^c_{ij}} (\eta^+_j(B^{sc}_i(q)) C^+_{jk}) + \sum_{k \in \mathcal{S}^{crit}_{ij}} C^+_{jk} \right) \quad (10)$$

Here, the only difference between Eq. 7 and Eq. 10 is that $\mathcal{D}^{sc}_{ij}$ has been replaced with $\mathcal{S}^{crit}_{ij}$.

VII. Response-time analysis for asynchronous task chains

In this section, we leverage our analysis approach for synchronous task chains in order to improve the latency estimates for conventional (asynchronous) chains as well. In contrast to the synchronous case, an asynchronous task chain i may be subject to a pipelined, i.e. interleaved, execution, depending on the input event model ($\boldsymbol{\eta}_i$). As multiple asynchronous activations of higher priority tasks can potentially queue up, we need to reconsider the impact of deferred activations.

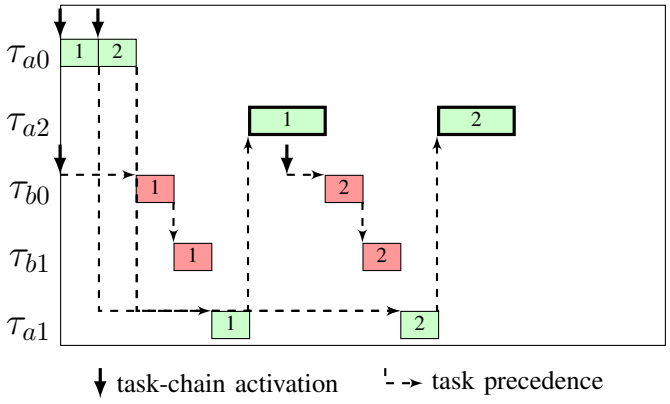


Figure 6. Gantt chart for an interleaved execution of task chain a and deferred asynchronous activations of τ_{a2} (bold).

Figure 6 illustrates such an interleaved execution of task chain a from our example in Figure 4. First, there are two consecutive activations of task chain a that result in τ_{a0} being executed twice. In turn, this releases two activations of τ_{a1}, which, however, are blocked by the pending activation and execution of task chain b. Thus, after the completion of chain b, τ_{a1} executes for the first time and eventually releases τ_{a2}. In the meantime, chain b is activated again and executes after τ_{a2}'s completion. The second activation of task chain a can only complete after b is idle again.

In contrast to the synchronous case (cf. Eq. 5), a task can only be considered as a deferred task if it has a predecessor whose execution is blocked. I.e. the first tasks of an asynchronous task chain (here: τ_{a0}) can still interfere arbitrarily given they have a higher-priority than any task of the analysed task chain.

Definition 9. *The set of deferred tasks of an asynchronous task chain j w.r.t. a task chain i is defined as follows:*

$$\forall j \neq i : \quad \mathcal{D}^{ac}_{ij} = \{k \in \mathcal{H}^c_{ij} | \exists l < k : p_{jl} < \min_m p_{im}\} \quad (11)$$

The definition of deferred tasks is essentially the only modification that is required to apply the (simple) task-chain busy window of Eq. 7 to the asynchronous task chains. By looking at the Gantt chart, we presume that although there can be multiple asynchronous activations of a deferred task

(τ_{a2}), only a single activation can interfere with our analysed task chain.

Theorem 2. *A deferred task $\tau_{jk} \in \mathcal{D}_{ij}^{ac}$ of an asynchronous task chain j can only interfere once with task chain i.*

Proof. By definition, a deferred task τ_{jD} always has a lower-priority predecessor τ_{jL}. Hence, according to the preemptive static-priority scheduling, a deferred activation of a τ_{jD} will always execute before τ_{jL}. Consequently, two executions of τ_{jL} are always separated by an execution of τ_{jD}. Therefore, at most one activation of τ_{jD} can be pending at any point in time. Furthermore, as τ_{jL} can never execute while task chain i is busy, τ_{jD} can only interfere once within i's busy window. □

Corollary 3. *The q-event busy window for an asynchronous task chain i is constructed by including the intra-chain interference and by replacing $\mathcal{I}_{ij}^{c}$ and $\mathcal{D}_{ij}^{sc}$ in Eq. 7 with their corresponding definitions for asynchronous chains:*

$$B_i^{ac}(q) = \sum_k \max(\eta_i^+(B_i^{ac}(q)), q) C_{ik}^+ + \sum_{j \neq i} \left(\sum_{k \in \mathcal{I}_{ij}^c} (\eta_j^+(B_i^{ac}(q)) C_{jk}^+) + \sum_{k \in \mathcal{D}_{ij}^{ac}} C_{jk}^+ \right) \tag{12}$$

with $\mathcal{I}_{ij}^{c} = \mathcal{H}_{ij}^{c} \setminus \mathcal{D}_{ij}^{ac}$.

However, due to the potentially interleaved execution of an asynchronous task chain, the refined approach that we presented for the synchronous case cannot be applied here, i.e. the interference from a deferred segment does not mutually exclude the interference from another deferred segment of the same asynchronous task chain.

VIII. Evaluation

In this section, we first provide a detailed experimental comparison of the analyses presented in Section VI and show its improvement over conventional CPA. In addition, we demonstrate how our analysis approach enhances the applicability of (automated) timing verification by means of an automotive use case.

A. Experimental results

In order to evaluate our analysis approach in general, we performed three experiments for which we compared the resulting latency bounds of different analyses.

All **three experiments** comprise two task chains containing six tasks in total that are mapped to the same resource (similar to Figure 4). For the different experiments, we changed the structure of the task chains as illustrated in Figure 7.

We further assigned six distinct priorities to the tasks and performed our analyses for all 720 permutations. Note that we selected this generalised (task-level) priority assignment as opposed to a thread-level assignment due to the fact that the latter would heavily influence the convergence of the CPA (cf. Section VIII-B) and thus limit the comparability with our approach. Table I specifies the worst- and best-case execution times of the tasks. Moreover, the task chains were activated by periodic input event models as specified in Table II.

Table I
Tasks used in the experiments and their core execution times

Task(s)	τ_{a0}	τ_{a1}	τ_{a2}	τ_{b0}	τ_{b1}, τ_{a4}	τ_{b2}, τ_{a3}
WCET	10	2	4	3	9	5
BCET	1	2	2	1	4	3

Table II
Input event models of task chain a and b used for the experiments

Experiment	Period a	Jitter a	Period b	Jitter b
3:3	20	5	100	0
4:2	30	5	100	0
5:1	40	5	100	0

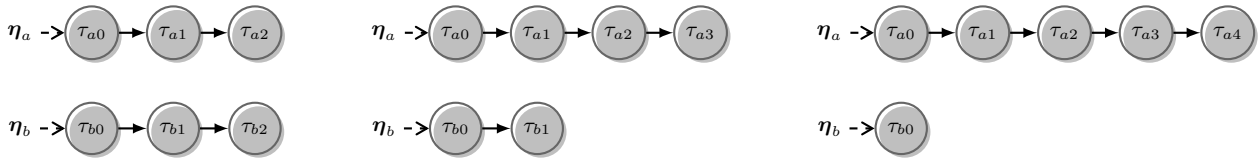
In each experiment, we performed the following **four analyses** for every priority assignment:

1) Conventional CPA with subsequent path analysis to derive the latency bounds for the task chains (cf. Section III).
2) WCRT analysis for synchronous task chains based on the simple task-chain busy window (Eq. 7).
3) WCRT analysis for synchronous task chains based on the refined task-chain busy window (Eq. 10).
4) WCRT analysis for asynchronous task chains based on the task-chain busy window as in (Eq. 12).

All analyses have been performed with a modified version of pyCPA [21] that implements our analysis approach. We limited the number of fixed-point iterations in Eq.1 to 1000, in order to catch the non-convergence case of the CPA.

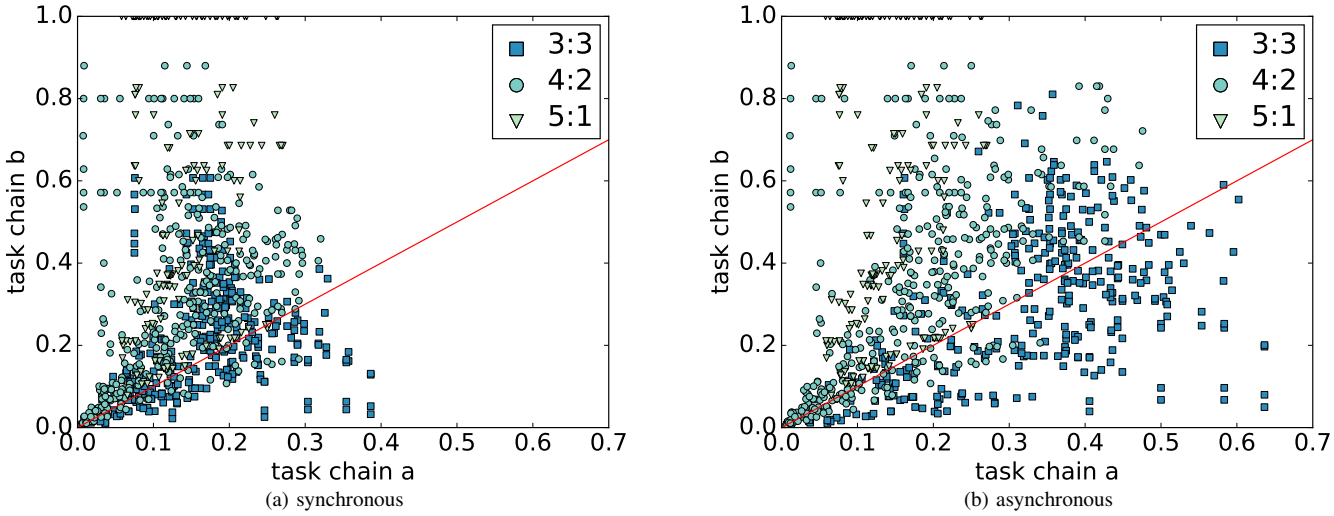
The results are summarised in Table III. While our analyses (2)-(4) successfully terminated in all cases, a substantial number of conventional CPA runs failed because it did not reach convergence within the limited number of iterations. Note that this is already a significant improvement over other approaches that still rely on a WCRT analysis of the single tasks, such as [14]. In all remaining cases, the analyses (2)-(4) provided better latency bounds for both task chains in the first two experiments. In the third experiment (5:1), this also holds for task chain a whereas the latency bound for task chain b improved only in 125 of 180 cases and showed the same result as the conventional CPA for the other 55 priority assignments. This is explained by the fact that task chain b only contains a single task in the third experiment and thus can only improve on the inter-chain interference, i.e. only for priority assignments for which τ_{b0} interferes with at least one but not all tasks of chain a.

In more detail, Figure 8a and 8b depict scatter plots of the relative improvements on the latency bounds from our analyses (2) and (4) over conventional CPA respectively. The relative improvement is determined by dividing the new result by the result from the conventional CPA. Here, the x-axis determines the improvement for task chain a whereas the y-axis shows the same for task chain b. Different markers are used in order to



(a) 3:3 experiment (b) 4:2 experiment (c) 5:1 experiment

Figure 7. Task chains used for the different experiments.



(a) synchronous (b) asynchronous

Figure 8. Relative latency improvements by our analyses for synchronous and asynchronous task chains compared to conventional CPA.

Table III
RESULT SUMMARY

exp.	# runs	# failed (conventional)	# improved (task chain a)	# improved (task chain b)
3:3	720	366	354	354
4:2	720	359	361	361
5:1	720	540	180	125

differentiate the experiments. Furthermore, we added a line for orientation that indicates the points where both chains would experience the same relative improvement.

We recognise that the results basically accumulate around this line, demonstrating a similar improvement for many cases. We also observe that analysis (2) and (4) show at least a relative improvement of 0.4 and 0.7 respectively for task chain a. Moreover, as the length of task chain a is increased (and task chain b is shortened in return), the latency bound improves even more for task chain a while this effect gradually decreases for chain b. For 55 priority assignments in the 5:1 experiment, the latency bound could not be improved at all, as the markers on the very top the plots indicate. Hence, as expected, the potential of improving the latency bound with our analysis approach correlates with the length of the analysed task chains.

We also compared the results from the simple and refined

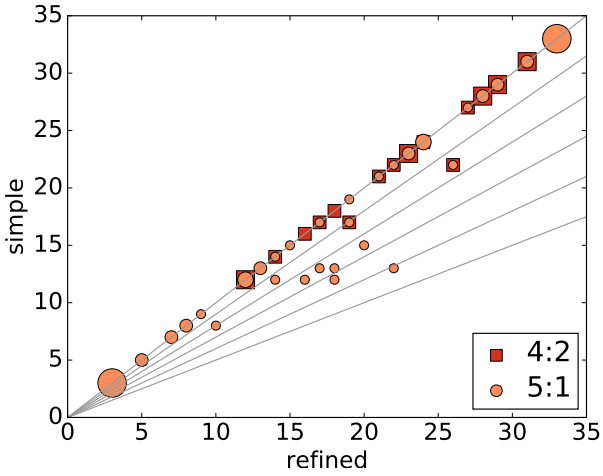


Figure 9. Resulting latency bounds for task chain b from our refined analysis approach compared to our simple approach.

approach, i.e. analyses (2) and (3), in the last two experiments for all 720 cases. Here, the refined approach improved for chain b in 48 (120) cases in the 4:2 (5:1) experiment respectively. Note that due to the length of the task chains in the 3:3 experiment, there can only be one deferred segment and thus no improvement over the analysis (2). Figure 9 depicts

the detailed results of this improvement by comparing the end-to-end latency bound for task chain b from the refined analysis (Eq. 10) on the x-axis with the results from the simple approach (Eq. 7) on the y-axis. For orientation, we added helper lines that indicate the 0%, 10%, 20%, 30%, 40% and 50% improvement. Note that the size of the markers corresponds to the number of analysed cases with the same results.

B. Analysis of an automotive use case

In this section, we demonstrate the applicability of our approach to an automotive use case that has been developed in the scope of the research unit *Controlling Concurrent Change (CCC)*, where a contract-based process is currently being implemented that enables the automated (in-system) integration of software components with non-functional constraints [28].

This use case implements a park and lane-assist function and involves several software components (i.e. threads) for *trajectory calculation* (TC), *object recognition* (OR1 and OR2), *object masking* (OM) and *steering* (S) required by the higher-level components that eventually implement the *parking assistant* (P) and *lane detection* (L).

Figure 10 depicts the thread communication of this use case. The obligation of the contracting process is to find a priority assignment that satisfies the given latency constraints. The task graph corresponding to this thread communication is illustrated in Figure 11. This task graph comprises two synchronous task chains P and L, one for the park assist and another one for the lane assist function respectively. The figure also indicates the threads from which the tasks originate. Note that the tasks inherit their priority from the corresponding seven threads. We do not assign the same priority to multiple threads as this typically only adds more pessimism to the analysis result.

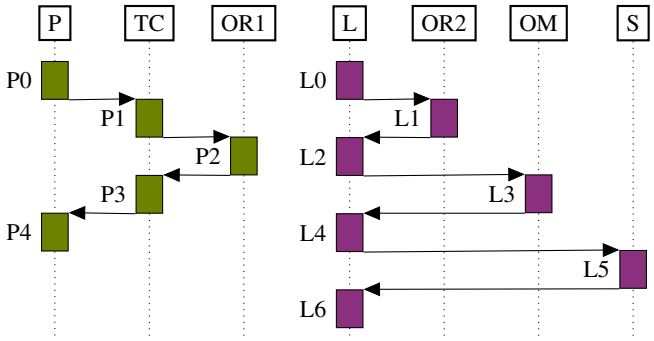


Figure 10. Thread communication for the park and lane assist use case.

In order to show the applicability and benefit of our analysis approach in the scope of this use case, we determined the solution space that we face depending on the implemented timing analysis. For this purpose, we performed a conventional CPA as well as our refined analysis for synchronous task chains for all possible priority assignments in the given scenario.

Table IV specifies the worst-case and best-case execution times of the tasks on which we based the analyses. As input event models, we assumed a period of 200 ms for chain P and a period of 100 ms for chain L as well as a jitter of 5 for both. According to the use case, the maximum acceptable latency for both task chains is 150 ms.

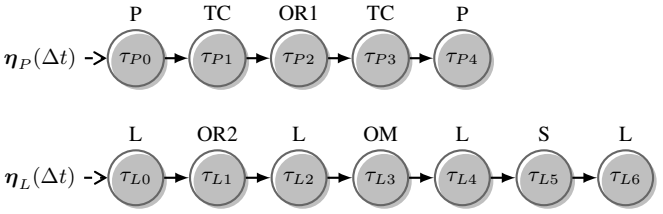


Figure 11. Task graph of a potential configuration for the park and lane assist use case. The labels above the tasks indicate from which software component (here: thread) the task originates.

Table IV
SPECIFICATION OF WORST-CASE (WCET) AND BEST-CASE EXECUTION TIMES (BCET) [IN MILLISECONDS] OF THE TASKS IN FIGURE 11

	τ_{P0}	τ_{P1}	τ_{P2}	τ_{P3}	τ_{P4}		
WCET	3	5	50	5	7		
BCET	1	1	10	1	1		
	τ_{L0}	τ_{L1}	τ_{L2}	τ_{L3}	τ_{L4}	τ_{L5}	τ_{L6}
WCET	3	10	3	10	10	10	4
BCET	1	5	1	5	5	5	1

Note that we reduced the maximum number of fixed-point iterations to 100 in order to keep the computational effort of the conventional CPA tractable. As a result, performing the conventional CPA for all 5040 priority assignments took about eight hours on single core of a conventional desktop CPU whereas our analysis approach only required 22 seconds. Moreover, the former reached the maximum number of iterations in all but 6 cases and therefore determined the priority assignment not schedulable. For the remaining analysable priority assignments it achieved latency results between 4949 and 8613 ms for chain P and between 1017 and 2322 ms for chain L. Thus, by means of conventional CPA, there is no feasible priority assignment.

On the contrary, our analysis approach draws a totally different picture, which is illustrated by the scatter plot of the resulting latencies for both task chains in Figure 12. In comparison, we also performed an offset-based analysis as presented in [18] with MAST [23]. Both analyses returned latency results for all priority assignments and below those of conventional CPA. The lines in Figure 12 mark the latency constraints, i.e. the space of acceptable solutions is below and left of these lines, which contains 2880 (11) of the 5040 priority assignments for our approach (MAST). The size of the markers corresponds to the number of analysed cases with the same results.

IX. CONCLUSION AND FUTURE WORK

In this paper, we presented an approach to end-to-end latency analysis for tasks chains that significantly reduces the pessimism of the worst-case timing analysis. From the model transformation between the programming model and the timing analysis model we derived that threads naturally

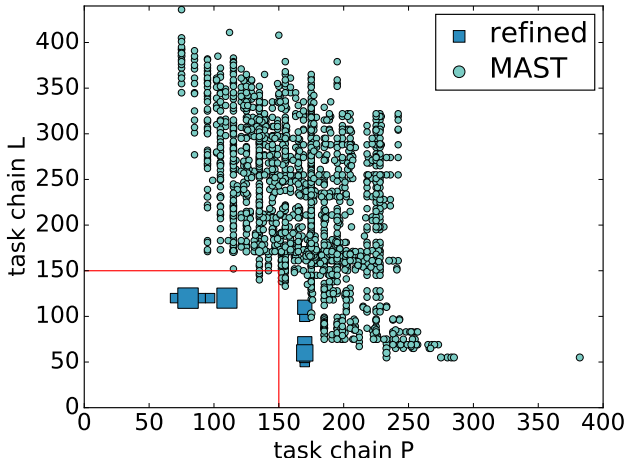


Figure 12. Resulting worst-case latency bounds [in ms] from the refined analysis and MAST. The lines indicate the acceptable solution space.

build chains of tasks with precedence constraints, which motivates the high relevance of this analysis. Furthermore, as these precedence relations affect the actual execution order (scheduling), we believe that the analysis of such chains are best approached by local scheduling analysis. Based on the busy-window approach that is typically applied on the task level, we derived a response-time analysis for entire task chains. We could not only show that this approach is able to significantly improve the end-to-end latency bounds but also increases the number of analysable systems as it reduces the number of fixed-point iterations. In contrast to other approaches, we were therefore able to improve the results (bounds) from the timing analysis while simultaneously reducing the computational effort. Consequently, this enhances the applicability and even enables the in-field use of timing analysis for common scenarios, which we also demonstrated by means of an automotive use case that has been developed in the scope of the CCC project. In future work, we will address how this approach can be extended to allow joins in task chains as this is a case we face as soon as software components (e.g. a communication stack) are shared among several components.

Acknowledgements

This work was supported by the DFG Research Unit Controlling Concurrent Change (CCC), funding number FOR 1800. We thank the members of CCC for their support.

References

[1] M. Stigge, "Real-time workload models: Expressiveness vs. analysis efficiency," Ph.D. dissertation, Uppsala University, 2014.
[2] AUTOSAR website. [Online]. Available: http://www.autosar.org/
[3] PikeOS Hypervisor. [Online]. Available: https://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept/
[4] seL4 Microkernel. [Online]. Available: https://sel4.systems/
[5] S. Chakraborty, S. Kunzli, and L. Thiele, "A general framework for analysing system properties in platform-based embedded system designs," in *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, ser. DATE '03. Washington, DC, USA: IEEE Computer Society, 2003.
[6] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System Level Performance Analysis - the SymTA/S Approach," in *IEEE Proceedings Computers and Digital Techniques*, 2005.
[7] R. Henia, L. Rioux, N. Sordon, G.-E. Garcia, and M. Panunzio, "Integrating Formal Timing Analysis in the Real-Time Software Development Process," in *Proceedings of the 2015 Workshop on Challenges in Performance Methods for Software Development*, ser. WOSP '15. New York, NY, USA: ACM, 2015, pp. 35–40.
[8] K. Richter, "Compositional Scheduling Analysis Using Standard Event Models," Ph.D. dissertation, TU Braunschweig, Braunschweig, Germany, 2005.
[9] J. Diemer, P. Axer, and R. Ernst, "Compositional Performance Analysis in Python with pyCPA," in *3rd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, Jul. 2012.
[10] S. Schliecker, J. Rox, M. Ivers, and R. Ernst, "Providing Accurate Event Models for the Analysis of Heterogeneous Multiprocessor Systems," in *Proc. 6th International Conference on Hardware Software Codesign and System Synthesis (CODES-ISSS)*, Atlanta, GA, Oct. 2008.
[11] J. Sun and J. W. S. Liu, "Bounding the end-to-end response time in multiprocessor real-time systems," in *Proceedings of the 3rd Workshop on Parallel and Distributed Real-Time Systems*, ser. WPDRTS '95. Washington, DC, USA: IEEE Computer Society, 1995.
[12] J. Lehoczky, "Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines," *Proc. 11th RTSS*, pp. 201–209, Dec 1990.
[13] K. Tindell, A. Burns, and A. Wellings, "An extendible approach for analyzing fixed priority hard real-time tasks," *Real-Time Systems*, vol. 6, no. 2, pp. 133–151, 1994.
[14] S. Schliecker and R. Ernst, "A recursive approach to end-to-end path latency computation in heterogeneous multiprocessor systems," in *Proc. 7th International Conference on Hardware Software Codesign and System Synthesis (CODES-ISSS)*. Grenoble, France: ACM, oct 2009.
[15] K. Tindell, "Adding time-offsets to schedulability analysis," Univ. of. York, UK, Tech. Rep. YCS 221, 1994.
[16] J. C. Palencia and M. González Harbour, "Schedulability analysis for tasks with static and dynamic offsets," in *Proceedings of the IEEE Real-Time Systems Symposium*, ser. RTSS '98. Washington, DC, USA: IEEE Computer Society, 1998.
[17] J. Mäki-Turja and M. Nolin, "Efficient implementation of tight response-times for tasks with offsets," *Real-Time Systems*, vol. 40, no. 1, pp. 77–116, 2008.
[18] J. C. Palencia and M. G. Harbour, "Exploiting precedence relations in the schedulability analysis of distributed real-time systems," in *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE*, 1999, pp. 328–339.
[19] R. E. Rafik Henia, "Improved offset-analysis using multiple timing-references," in *Proceeding Design Automation and Test in Europe*, Mar. 2006.
[20] K. Tindell and J. Clark, "Holistic schedulability analysis for distributed hard real-time systems," *Microprocess. Microprogram.*, vol. 40, no. 2-3, pp. 117–134, Apr. 1994.
[21] pyCPA website. [Online]. Available: https://bitbucket.org/pycpa/pycpa
[22] E. Wandeler and L. Thiele, "Real-Time Calculus (RTC) Toolbox," 2006. [Online]. Available: http://www.mpa.ethz.ch/Rtctoolbox
[23] MAST: modeling and analysis suite for real-time. [Online]. Available: http://mast.unican.es/
[24] SymTA/S: symbolic timing analysis for systems. [Online]. Available: https://www.symtavision.com/symtas.html
[25] S. Perathoner, "Modular performance analysis of embedded real-time systems: improving modeling scope and accuracy," Ph.D. dissertation, ETH Zurich, 2011.
[26] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Trans. Comput.*, vol. 39, no. 9, pp. 1175–1185, Sep. 1990.
[27] U. Steinberg, A. Böttcher, and B. Kauer, "Timeslice Donation in Component-Based Systems," in *6th OSPERT*, Brussels, Belgium, 2010.
[28] J. Schlatow, M. Moestl, and R. Ernst, "An extensible autonomous reconfiguration framework for complex component-based embedded systems," in *12th International Conference on Autonomic Computing (ICAC 2015)*, Grenoble, France, July 2015, pp. 239–242.