Response-Time Analysis for Task Chains with Complex Precedence and Blocking Relations

JOHANNES SCHLATOW and ROLF ERNST, Institute of Computer and Network Engineering, TU Braunschweig, Germany

For the development of complex software systems, we often resort to component-based approaches that separate the different concerns, enhance verifiability and reusability, and for which microkernel-based implementations are a good fit to enforce these concepts. Composing such a system of several interacting software components will, however, lead to complex precedence and blocking relations, which must be taken into account when performing latency analysis. When modelling these systems by classical task graphs, some of these effects are obfuscated and tend to render such an analysis either overly pessimistic or even optimistic.

We therefore firstly present a novel task (meta-)model that is more expressive and accurate w.r.t. these (functional) precedence and mutual blocking relations. Secondly, we apply the busy-window approach and formulate a modular response-time analysis on task-chain level suitable but not restricted to static-priority scheduled systems. We show that the conjunction of both concepts allows the calculation of reasonably tight latency bounds for scenarios not adequately covered by related work.

CCS Concepts: • **Computer systems organization → Real-time systems**; *Embedded and cyber-physical systems*;

Additional Key Words and Phrases: response-time analysis, component-based software systems, service-oriented architectures

ACM Reference format:
Johannes Schlatow and Rolf Ernst. 2017. Response-Time Analysis for Task Chains with Complex Precedence and Blocking Relations. *ACM Trans. Embed. Comput. Syst.* 16, 5s, Article 172 (September 2017), 19 pages.
https://doi.org/10.1145/3126505

1 INTRODUCTION

In the following pages, we address the response-time analysis (RTA) of complex software systems. Those systems often resort to component-based approaches that not only benefit the development process but also improve the isolation and therefore security, especially when combined with a microkernel architecture. For instance, we notice a trend towards these architectures in the automotive domain in order to deal with the increasing complexity of advanced driver assistance systems and automated driving functions. In particular, such architectures are already commercially available for this domain (e.g., QNX Neutrino [2]). This design methodology is focused on the communication between software components such that a single functionality is therefore implemented by multiple interacting software components. More specifically, service-oriented approaches are often applied in which a single instance of a software component (microserver) may be used in different contexts. When modelling these systems for timing analysis, we not only have

Authors' addresses: J. Schlatow and R. Ernst, TU Braunschweig, Institute of Computer and Network Engineering, Hans-Sommer-Straße 66, 38106 Braunschweig, Germany; emails: {schlatow, ernst}@ida.ing.tu-bs.de.

https://doi.org/10.1145/3126505

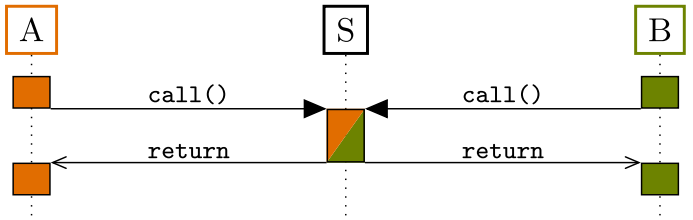

Fig. 1. Sequence diagram for the microserver (S) example with two clients (A, B).

to consider precedence relations that result from the communication but also blocking relations that originate from shared services. Moreover, from a functional perspective, latency constraints are rather formulated for entire task chains than a single task.

As our core contribution, we therefore present an RTA on task-chain level which covers the above scenarios and that is based on the busy-window approach for single static-priority scheduled processors. For this purpose, we additionally introduce an abstract task model that expresses the precedence and blocking relations that result from shared services and different communication semantics. With this, we try to close the gap between very powerful but complex timing-aware modelling tools – such as MARTE UML [3] – and very basic models, which simplify or neglect these aspects but still receive a lot of attention in research.

The following section moves on to explain the scope of our work in greater detail by introducing a motivational example. In Section 3, we introduce and explain our model that particularly addresses the complex precedence and blocking relations of our targeted software systems. Based on this, we present our RTA for the latter in Section 4 before we summarise the related work in Section 5. This is followed by an experimental evaluation in Section 6 and our conclusion in Section 7.

2 MOTIVATIONAL EXAMPLE

As introduced in the previous section, we particularly focus on the RTA for microkernel-based systems. Those systems typically host multiple software components (processes) that implement e.g., applications, device drivers or protocol stacks. Inter-component communication (or inter-process communication (IPC)) enables these components to interact and eventually implement the desired functionality of the (embedded) system. The explicit nature of this communication exposes the functional chains, which can be subject to real-time constraints. It is therefore essential to model and analyse this communication adequately in order to derive formal upper bounds on the response times.

In microkernel-based (and also in component-based) systems, we distinguish between *synchronous calls* and *asynchronous messaging*. While the former has blocking semantics, i.e., the caller waits for the reply of the callee, the latter is non-blocking and hence allows concurrency. Synchronous IPC gained popularity in L3/L4-based microkernels as it significantly increased the performance of the microkernel approach [12]. Nevertheless, modern L4-based microkernels also provide mechanisms to efficiently exchange asynchronous notifications (much like software interrupts) [7].

Because of the mentioned popularity of synchronous IPC in microkernels, supposedly the most commonly used scheme is the *microserver* scheme in which a component provides a service to multiple other components via a remote procedure call (RPC) interface. Figure 1 depicts such a microserver component that is synchronously called by two different client components. Although the procedure call semantics of such an interface is a very natural abstraction, it comes with the drawback of inter-dependent blocking effects if the server component is single-threaded.

While the blocking effects in the simple microserver scenario are relatively easy to grasp, it is also common to use more complex communication schemes between components by combining synchronous calls with asynchronous notifications [8, Section 3.6]. Hence, the functional chains will comprise different communication semantics and potentially hidden blocking effects that depend on the scheduling of the underlying kernel. (In Section 3, we will introduce a concrete example for such a scenario.) We therefore developed a systematic approach to model these effects as an abstraction of the actual kernel implementation. Before we go into detail, let us first summarise what priority-based scheduling policies can be found in prominent microkernels that we would like to cover with our approach.

As soon as blocking comes into play, priority inversion is a well-known effect that must be taken into consideration [22]. Priority inversion is typically mitigated by priority inheritance, which can be found in microkernel implementations, e.g., as so-called time slice donation and helping in NOVA [23] and Fiasco.OC, thread migration in Composite [15], and a similar mechanism in the QNX Neutrino RTOS [2]. The underlying principle of these mechanisms is that a caller contributes its scheduling context (i.e., its priority, time slice or thread) to the callee so that the latter inherits the scheduling parameters of all waiting callers. It is our understanding that a timing model for such systems should explicitly capture these effects (inversion and inheritance) and thereby generalise from the kernel implementation. In the scope of this paper, we develop a single-processor timing model that achieves this. Although our analysis focuses on typical priority-based scheduling found in modern microkernels, our modelling approach is applicable to multi-threaded systems with explicit communication (e.g., service-oriented architectures) in general.

3 EFFECT-ORIENTED MODELLING

In this section, we develop a software timing model for software systems of communicating software components which explicitly models blocking and precedence relations that result from the communication semantics and scheduling features. Given sufficient knowledge about the operating system (OS), such a platform-specific timing model can be translated, e.g., from more general sequence diagrams. Note that the scope of our timing model is a single processing resource and we assume the system is free of deadlocks.

The software timing is typically modelled in *task graphs*, which express the precedence relations (directed edges/arcs) and execution times of different tasks (vertices), and serves as a basis for schedulability and response-time analysis. In particular, an outgoing arc in such a task graph means that a new job of the target task is released every time a job of the source task finished. Annotating the tasks with best-case and worst-case execution times then allows performing a conservative timing analysis.

A task graph, or timing model in general, should describe precedence and concurrency in a way that allows a timing analysis to accurately bound possible interference. In order to specify an accurate task graph for a given software system, a deep understanding of the interacting software components is therefore necessary. As shown in Figure 1, sequence diagrams are a natural way to describe this inter-component communication and have already been used for extracting task graphs by associating each activity with a task and transferring the already described precedence relations. In particular, Henia et al. [9] have shown how synchronous calls and asynchronous notifications are reflected in the task graph, which closes the semantic gap between both modelling approaches.

Yet, if applied to our microserver example in Figure 1, the resulting task graph obfuscates the (functional) event chains. This is illustrated by Figure 2, which depicts a naive translation of the five activities into tasks and their precedence relations. The task τ_S represents the activity of the shared microserver, which is activated by either of the client tasks τ_{A1} or τ_{B1}. By only looking at

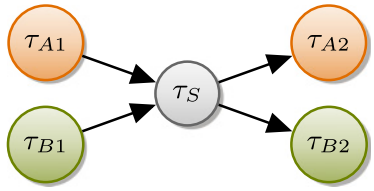

Fig. 2. Naive task graph representation of Figure 1.

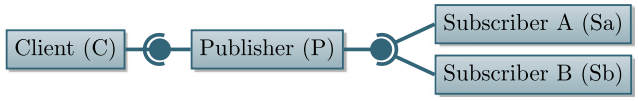

Fig. 3. Illustrative use case combining client-server and publisher-subscriber communication.

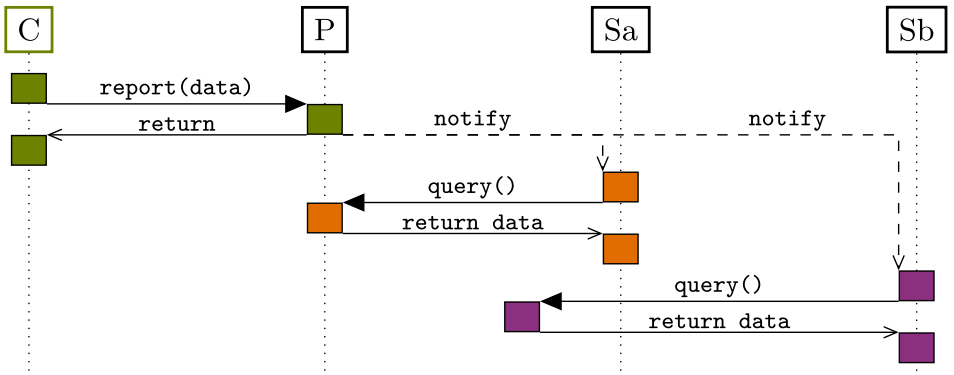

Fig. 4. Sequence diagram for Figure 3.

the task graph, however, we typically assume that both tasks, τ_{A2} and τ_{B2}, are activated for every activation of τ_S (fork semantics). This obfuscates the modelled software system in which the server S only replies to the actual caller (not both). Although similar scenarios have been addressed by Perathoner et al. [16] with structured/hierarchical event streams, this approach still requires background knowledge not explicitly expressed in the task graph. Alternatively, our key observation is that the same scenario can be modelled as distinct task chains which, however, are subject to mutual blocking at the server task(s). Thus, our modelling approach consists in preserving the event chains while explicitly expressing effects such as blocking and priority inheritance. Before proceeding to the description of our task model that we developed for this purpose, let us first introduce a more elaborate use case that we will use for illustration.

3.1 Illustrative Use Case

Our illustrative use case is depicted in Figure 3 and comprises four single-threaded components. The central component (publisher P) acts as a microserver and provides two services: the client service and the subscriber service. The sequence diagram in Figure 4 details the inter-component communication in this scenario: The client component C reports data to the publisher which notifies the subscribers Sa and Sb of newly available data. The subscribers can then use the corresponding service to query the actual data from the publisher P. The result are three synchronous transactions that a) experience blocking at the publisher component and b) build two event chains

(from the client to each subscriber). Furthermore, we observe that synchronous transactions enforce a strict order of the activities due to the fact that the caller (e.g., C) waits for the return of the callee (P) and can therefore not be re-entered before the entire transaction is completed. On the other hand, we observe that asynchronous notifications do not result in a particular order between the transactions. In our example, this means that the transactions of Sa and Sb may interfere with each other and can also be interrupted by a new transaction of C (depending on the scheduling policy). In addition, the blocking at P results in the fact that its activities cannot preempt each other.

Let us capture these observations by formulating two different semantics of precedence relations that we later formalise:

(1) *Strict precedence* results from synchronous/blocking communication that causes a fixed order between the strictly preceding elements in any execution trace.
(2) *Weak precedence* results from asynchronous/non-blocking communication and causes a sequential order between the preceding elements but allows an interleaved execution of multiple sequences.

In this paper, we illustrate strict precedence with solid arrows and weak precedence with dashed arrows.

3.2 Task Model

In this section, we generalise the observations motivated by the examples described above in order to formulate a more expressive task model w.r.t. platform and OS aspects. Let us first highlight what questions arise when performing an (accurate) schedulability and RTA in order to derive the requirements for our model:

What scheduling policy is implemented? The most prominent scheduling policies for real-time systems are static-priority (non-)preemptive scheduling (SP(N)P) and earliest deadline first (EDF). The scheduling policy models the decision-making of the system's scheduler that assigns the processing time of the CPU(s) to the different threads. A RTA must be aware of this policy in order to derive accurate bounds on the interference that a thread might experience from other threads. The task model itself should be agnostic of the scheduling policy w.r.t. how scheduling decisions are made but expose where/when these decisions are made.

What is the scheduled entity? In contrast to what a conventional task graph indicates, tasks are seldom the entities scheduled by an operating system; or to put this in another way: the entities scheduled by an OS seldom resemble a single task that only communicates or interacts with the OS at the beginning and end of its execution. Our task model shall therefore explicitly describe to which scheduled entity a task belongs. For this, we introduce the concept of *scheduling contexts*, which are scheduled if there is work (i.e., a job of a task) ready to be executed within this context.

How do scheduling parameters propagate during communication? As explained earlier, synchronous IPC is best combined with a (priority-)inheritance mechanism that prevents inversion effects. From a more abstract perspective, we see this as a propagation/inheritance of scheduling parameters. In our model, these parameters are attributed to the scheduling contexts. The mapping of tasks to these scheduling contexts thus exposes with what scheduling parameters a task is executed.

When can components be re-entered? This question addresses the blocking relations already pointed out in our use case. In summary, these result from the fact that a component not only needs to wait for the return of a callee but must also reach a point in its code where it is ready to receive calls or notifications on its own. As our task model shall include these effects, we introduce the concept of *execution contexts* that resemble shared resources. In particular, a task (i.e., its job) can only be executed if its execution context is not already used by another task.

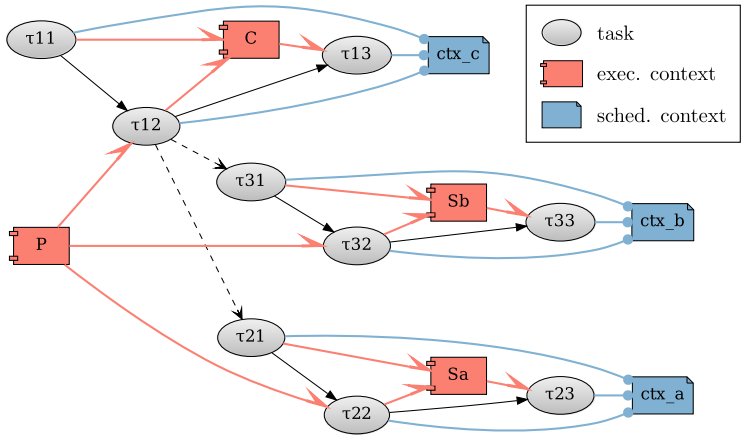

Fig. 5. Extended task model for our use case in Figure 4.

Having stated the key aspects that our model shall incorporate, we now present its formal definition.

Definition 3.1. (task model)
The *task model* is a graph $G = (V_t, V_e, V_s, A_t, A_e, E_s)$ with

- three disjoint sets of vertices (set of tasks V_t, set of execution contexts V_e, set of scheduling contexts V_s) and
- three disjoint sets of arcs/edges: A_t, A_e, E_s,

which result from combining the following sub graphs:

(1) The directed acyclic *task graph* $G_t = (V_t, A_t)$, which describes the workload to be scheduled and its precedence relations.
(2) The bipartite directed *allocation graph* $G_e = (V_t, V_e, A_e)$, which describes the allocation of execution contexts by tasks.
(3) The bipartite *mapping graph* $G_s = (V_t, V_s, E_s)$, which describes the mapping of tasks to scheduling contexts.

Figure 5 depicts the task model that results from the sequence diagram in Figure 4. The tasks $\tau 11$, $\tau 12$, $\tau 13$ represent the transaction between C and P while $\tau 21$, $\tau 22$, $\tau 23$ and $\tau 31$, $\tau 32$, $\tau 33$ correspond to the transaction from Sa and Sb to P respectively. We elaborate on the semantics of the sub graphs in the following sections and describe how these are represented in the figure. For this, let $indeg(v)/outdeg(v)$ denote the in/out degree of a vertex v.

3.2.1 Semantics of the Task Graph G_t. The task graph is a directed graph which describes the precedence relations between tasks and hence models how the workload that needs to be scheduled on the processor is released (tasks and solid/dashed arrows in Figure 5). More precisely, A_t denotes the set of arcs as ordered pairs of $\tau \in V_t$ such that for every $(\tau_i, \tau_j) \in A_t$, a job of a task τ_j is released once a job of τ_i completed. In the scope of this paper, we restrict the task graph to be a "forest", i.e., each connected component of G_t is tree shaped, such that $\forall \tau \in V_t : indeg(\tau) \leq 1$. We observed that most of our inter-component communication scenarios break down to those tree-shaped graphs. However, this restriction needs to be relaxed as soon as synchronisation of multiple concurrent paths is required (e.g., voter scenarios), which is outside the scope of this paper.

Let us briefly clarify some terminology to which we later refer: The *activation* of a task denotes the release of a job. A task is *pending* if there is at least one unprocessed job. We further define *(direct) successors/predecessors* of a task.

Definition 3.2. The direct successors and predecessors of a task τ_i are defined as follows:

$$succ_i = \{\tau_j | \exists (\tau_i, \tau_j) \in A_t\} \tag{1}$$

$$pred_i = \{\tau_j | \exists (\tau_j, \tau_i) \in A_t\} \tag{2}$$

Furthermore, let $succ_i^*$ ($pred_i^*$) denote the set of directly or indirectly connected tasks τ_j, i.e., τ_j is reachable from τ_i (or vice versa).

In Figure 5, the direct predecessor and successor of $\tau 31$ are $\tau 12$ and $\tau 32$ respectively. Moreover, the (indirect) predecessors of $\tau 31$ are $\tau 11$ and $\tau 12$ whereas its successors are $\tau 32$ and $\tau 33$. Note that $\tau 12$ has multiple direct successors while it is not possible for any task to have multiple direct predecessors according to our definition.

3.2.2 Semantics of the Allocation Graph G_e. As indicated previously, the allocation graph describes how the execution of tasks blocks or releases a component. The set of arcs connect tasks with execution contexts (or vice versa), i.e., $A_e \subseteq V_t \times V_e \cup V_e \times V_t$. An incoming arc $(\tau, v) \in A_e$ to an execution context denotes that the execution context is allocated by the connected task and not released on completion (e.g., between $\tau 11$ and C in Figure 5). An outgoing arc $(v, \tau) \in A_e$, on the other hand, indicates the release of the execution context after the task's execution finished (e.g., between $\tau 12$ and P). We refer to both, allocation and release, as blocking. A task can only be scheduled if its execution context is not blocked or its predecessor already allocated this context. A valid allocation graph must adhere to the following conditions:

(1) A task can block multiple execution contexts.
(2) A task allocating an execution context must have exactly one direct successor that either allocates or releases the same execution context, i.e., $\forall \tau_i \in V_t, (\tau_i, v) \in A_e : \exists_1 \tau_j \in succ_i : (\tau_j, v) \in A_e \vee (v, \tau_j) \in A_e$.
(3) A task allocating an execution context must have a successor that releases the same execution context, i.e., $\forall \tau_i \in V_t, (\tau_i, v) \in A_e : \exists \tau_j \in succ_i^* : (v, \tau_j) \in A_e$.

Based on G_e, we distinguish strict/weak precedence as follows:

Definition 3.3. (strict precedence) An arc $(\tau_i, \tau_j) \in A_t$ is classified as *strict precedence* iff both tasks block the same execution context, i.e., $\exists v \in V_e : \exists \{\tau_i, v\} \in E_e \wedge \exists \{\tau_j, v\} \in E_e$, where E_e denotes the set of undirected edges from A_e. Analogously to Definition 3.2, $strpred_i^*/strsucc_i^*$ are defined as the set of tasks τ_j for which there exist a path to/from τ_i with only strict precedence arcs.

For instance, in Figure 5, strict precedence exists between $\tau 11$, $\tau 12$ and $\tau 13$. Note that $\tau 12$, representing a callee task in P, still allocates the execution context C and additionally blocks P which is, however, released upon the task's completion. The context C is eventually released after $\tau 13$. On the other hand, neither $\tau 21$ nor $\tau 31$ are strict successors as they do not block the same execution context as $\tau 12$ (emphasised by the dashed arrows in Figure 5).

3.2.3 Semantics of the Mapping Graph G_s. The edges in the mapping graph describe the unique mapping of tasks to scheduling contexts and thereby indicate when and based on what scheduling parameters a task is scheduled, i.e., $\forall v \in V_e : deg(v) = 1$. We refer to the edges $e \in E_s$ as unordered pairs $\{\tau, v\}$ with $\tau \in V_t$ and $v \in V_s$. A task which is mapped to a different scheduling context than

its direct predecessor must typically wait for an invocation of the scheduler and the decision to be scheduled. In Figure 5, we have assumed an inheritance scheme, which results in the callee tasks being mapped to the same scheduling context as the caller.

Note that a subset of our model can be expressed and analysed with related work, i.e., MAST [5]. Taking Figure 5 as an example, tasks relate to operations in MAST, scheduling contexts to scheduling servers and execution contexts to shared resources. In MAST, operations can lock/unlock a shared resource, however, a lock must not be hold across scheduling server boundaries This is in conflict with τ_{12}, which holds the lock of C when activating τ_{31} and τ_{21}.

3.3 Task Chains

The next part of this paper will present our RTA for task chains based on the model introduced above. Before proceeding to this part, let us briefly summarise how we define the task chains. While our task model specifies the entire workload of a single processor, the task chains specify the transactions for which the response-times shall be calculated.

Definition 3.4. (task chain) A task chain $\mathcal{T}_a$ is a sequence of directly connected tasks (on the same processor). For a task-chain response-time analysis, task chains can be specified arbitrarily as long as each task belongs to at least one chain.

Note that this definition also includes specifying a single task as a task chain. However, in order to benefit the most from our analysis approach, we typically specify a task chain for every sink (i.e., leaf in the task graph) such that it contains all predecessors (and the sink itself). In Figure 5, we thus find the following task chains: chain A from $\tau 11$ to $\tau 13$, chain B from $\tau 11$ to $\tau 23$, and chain C from $\tau 11$ to $\tau 33$.

4 RESPONSE-TIME ANALYSIS

In the following pages, we present our RTA which is based upon our modelling approach introduced above. Note that we assume static-priority (preemptive) scheduling. In order to perform this analysis, we annotate the tasks with best-case and worst-case execution times, and the scheduling contexts with a priority. Hence, let $\mathcal{L}_i$ denote the set of lower-priority, and $\mathcal{H}_i$ the higher and equal-priority tasks of τ_i. For readability, we also define the set of higher priority tasks for each task chain as follows:

Definition 4.1. $\mathcal{H}_a^c = \{\tau_j | \exists \tau_i \in \mathcal{T}_a : \tau_j \in \mathcal{H}_i\}$ denotes the set of higher or equal-priority tasks for task chain a.

For our RTA, we further assume the activation patterns for our input tasks (roots of the task graph) are given by a pair of arrival curves $\eta^+(\Delta t)/\eta^-(\Delta t)$ specifying an upper/lower bound on the number of events that can arrive within any half-open time interval $[t, t + \Delta t)$. Alternatively, the arrival curves can be expressed by their pseudo-inverse functions $\delta^-(n)/\delta^+(n)$ that specify the minimum/maximum time interval between any n consecutive events. We also base our analysis on the generalised busy-window technique [21] in order to calculate the busy window for every task chain on a processor (task-chain busy window), as we will detail in the next section. This provides us with an upper bound of the time the processor may take from the arrival of a particular input event to the production of a corresponding output event, i.e., the local response time. Note that this local RTA is an essential building block of a compositional performance analysis (CPA), which naturally deals with the analysis of distributed systems but has been shown to be overly pessimistic on the local response-times for event chains [19].

4.1 Task-Chain Busy Window

The task-chain busy window is the maximum time the processor may be busy processing the activations of a particular task chain. In general, a busy window must consider the following aspect:

(1) *Core execution time* is the time that is required to execute all parts of the task chain in isolation without any interference or blocking. This is a lower bound on the busy time.
(2) *Deferred load/blocking* is the workload or blocking time resulting from previous activations of other tasks (e.g., lower-priority blocking). It results from the fact that a busy window does not necessarily start when the processor is idle.
(3) *Incoming activations* are the cause of interference from arriving events within the busy window.

For this purpose, we need conservative and preferably tight upper bounds for the interference on a certain task chain. In the remainder of this section, we will formulate these bounds based on the model introduced earlier together with the arrival curves. Note that a tight timing analysis must incorporate assumptions about the (OS) implementation. However, changes of these assumptions shall not invalidate the entire approach. We therefore formulate the task-chain busy window as a collection of independently applicable bounds, which keeps this approach more modular.

Similar to [19], we define the q-event task-chain busy window for the chain under analysis (CUA) $\mathcal{T}_a$ as follows:

Definition 4.2. For any task chain $\mathcal{T}_a$, the q-event task-chain busy window $B_a^c(q)$ denotes the maximum time a processor may be busy processing q-events of $\mathcal{T}_a$, i.e., of all $\tau_i \in \mathcal{T}_a$. Furthermore, let Q_a denote the maximum q for which B_a^c is defined such that $\forall q > Q_a : \delta_a^-(q) > B_a^c(Q_a)$.

Note that the q-event busy window ends with the completion of the q-th activation of the last task in the CUA.

Lemma 4.3. *The maximum task-chain busy window, $B_a^c(Q_a)$, is the longest time the processor may take to re-transition into a state in which there are no pending activations of any $\tau_i \in \mathcal{T}_a$.*

Proof. We prove this by contradiction. Let us assume, there is a pending task $\tau_p \in \mathcal{T}_a$ after $B_a^c(Q_a)$. From Definition 4.2 we infer that the last task of the CUA executed Q_a times. Due to the precedence relations in the chain, all other tasks must have executed at least Q_a times. From the assumption, we infer that τ_p has a predecessor who executed $q > Q_a$ times. Applying the precedence relations, we know that the first task of the CUA also executed at least q times. Hence, this can only occur if $\delta_a^-(q) \leq B_a^c(Q_a)$, which contradicts the definition of Q_a. □

Based on this, we can calculate the worst-case response time of the CUA as follows:

Corollary 4.4. *As proved by Schliecker et al. [21], the worst-case response time is found among all q-event busy windows as follows:*

$$R_a^+ = \max_q (B_a^c(q) - \delta_a^-(q)) \tag{3}$$

We would like to emphasise these definitions do not guarantee that there are no pending activations of any $\tau_i \notin \mathcal{T}_a$ after $B_a^c(Q_a)$, potentially leading to deferred load. We approach this by systematically considering and bounding the different types of interference experienced by the CUA.

In order to calculate the $B_a^c(q)$, we distinguish between two semantics of bounding functions: event-count bounds and workload bounds. The former bound the number of activation events that can be seen for a certain task within the q-event task-chain busy window of the CUA. The

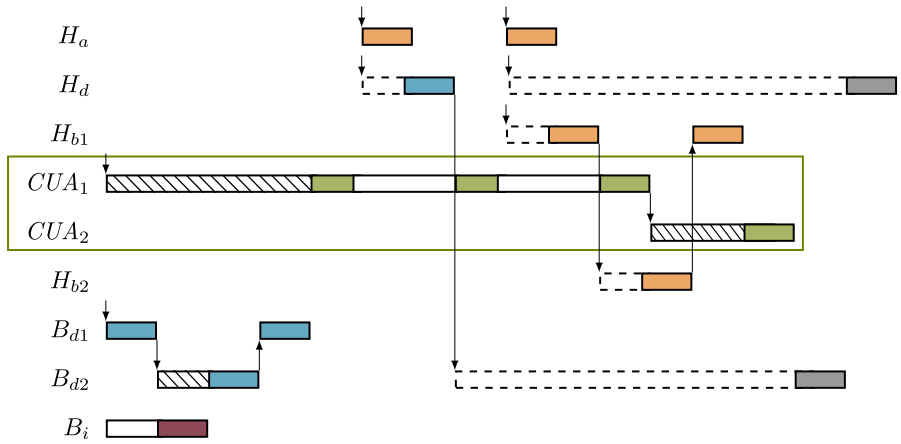

Fig. 6. Exemplary busy window for the chain under analysis CUA (green) with arriving higher-priority interference/blockers (H_a/H_b, orange), deferred interference/blockers (H_d/B_d, blue), indirect blockers (B_i, red), preempted execution (white), and blocked execution (hatched).

latter bound the workload that is induced by a certain task (within the particular busy window). Our framework for calculating the busy windows is based on using multiple of the above bounds under the following assumptions:

Definition 4.5. No event-count/workload bound used for calculating a busy window must be mutually exclusive with another bound, i.e., the selection of a particular bound must not invalidate another bound.

Definition 4.6. The event-count/workload bounds may depend on each others results, i.e., a bounding function may rely on and use the result of another bound.

Hence, we note that Definition 4.5 may significantly restrict the tightness of the bounds. Nevertheless, mutual exclusive bounds can still be applied in terms of a candidate search, which is, however, not in the scope of this paper. We also conclude that Definition 4.6 basically creates a fixed-point problem that requires propagating the intermediate results until the bounds converge.

In order to formulate such bounds, we further distinguish between two types of interference (cf. Figure 6):

Definition 4.7. The interference experienced by the chain under analysis is either accounted for as arriving or deferred interference. For arriving interference, we assume that there is no pending activation/workload at the start of the chain's busy window. On the other hand, deferred interference is based on pending activations/workload that may result from a previous busy window and is independent of the arrival curves.

We therefore must ensure that the arriving interference finishes within the q-event busy window of the CUA by including transitive interference effects.

Definition 4.8. Any bounding function must ensure that it safely bounds the interference that either the CUA or any arriving interferers may experience during the q-event busy window.

Before we elaborate on the particular bounds that we can derive from our model in the following sections, let us state our basic formula for $B_a^c(q)$.

Theorem 4.9. *The q-event task-chain busy window is computed as follows:*

$$\forall q \in [1, Q_a] : \quad B_a^c(q) = \sum_{\tau_i \in V_t} \min_k \theta_{a,i}^{(k)}(q) \tag{4}$$

where $\theta_{a,i}^{(k)}(q)$ denotes the k-th safe upper bound on the processing time (workload) required by τ_i while processing q events of $\mathcal{T}_a$.

Proof. Every bounding function safely bounds the interference from a particular task during the q-event busy window, respecting all possible interference scenarios (Definition 4.7 and Definition 4.8). By Definition 4.5, each bounding function must be valid at every time such that selecting the minimum is a safe operation even if we allow the bounds to depend on each others results (Definition 4.6). □

4.2 Workload Bounds

In the scope of this paper, we formulate a single workload bound per task that we calculate from an upper bound on the task's activations within a q-event task-chain busy window.

Definition 4.10. The workload from a task τ_i within the q-event busy window of task chain $\mathcal{T}_a$ is conservatively bounded by

$$\theta_{a,i}^{(1)}(q) = n_{a,i}(q) \cdot C_i^+ \tag{5}$$

where $n_{a,i}(q)$ denotes the safe upper bound for the number of activations of τ_i within the q-event busy window of $\mathcal{T}_a$.

Note that other and more elaborate workload bounds, e.g., by considering varying execution times [25], can be applied but are outside the scope of this paper. We therefore move on to detail how $n_{a,i}(q)$ can be bounded by event-count bounds.

4.3 Event-Count Bounds

In this section, we formulate lower and upper bounds on the number of activations that must be included in a q-event busy window of a task chain. First, let us have a look at the lower event-count bounds, which result from the fact that in any q-event busy window, every task $\tau_i \in \mathcal{T}_a$ must at least execute q times.

Definition 4.11. The lower event-count bound for a task τ_i within the q-event busy window of task chain $\mathcal{T}_a$ is given by $\zeta_{a,i}(q)$ and defined as follows:

$$\forall \tau_i \in \mathcal{T}_a : \zeta_{a,i}(q) = q \tag{6}$$

$$\forall \tau_i \notin \mathcal{T}_a : \zeta_{a,i}(q) = 0 \tag{7}$$

For the upper event-count bounds, we need to bound the interference that may be experienced when executing the q activations of the CUA. Note that every task in $\tau_i \in V_t$ may interfere with our CUA in the worst case, including the $\tau_j \in \mathcal{T}_a$ (self-interference). In the remainder of this section, we thus introduce several upper event-count bounds such that we can calculate $n_{a,i}(q)$ as follows:

Theorem 4.12. *The number of executions of τ_i in the q-event busy window of $\mathcal{T}_a$ is not greater than*

$$n_{a,i}(q) = \max\left(\zeta_{a,i}(q), \min_k \vartheta_{a,i}^{(k)}(q)\right) \tag{8}$$

where $\vartheta_{a,i}^{(k)}(q)$ denotes the k-th upper event-count bound for task τ_i in the q-event busy window of $\mathcal{T}_a$.

Proof. Similar to Theorem 4.9, we only rely on the validity of the lower and upper event-count bounds $\zeta_{a,i}(q)$ and $\vartheta_{a,i}(q)$ respectively. Due to Definition 4.5, every bound must be valid on its own, which allows us to apply the *min* and *max* operators. □

We conservatively start with an unbounded upper event-count function for every task.

Definition 4.13. The upper event-count bounds are initialised with the unbounded function $\vartheta^{(0)}_{a,i}(q)$:

$$\forall \tau_i : \vartheta^{(0)}_{a,i}(q) = \infty \tag{9}$$

Of course, this function alone will not allow us to compute a bounded busy window but ensures that every task will be considered unless it can be safely improved or excluded. For this purpose, we start with applying the arrival curves.

Theorem 4.14. *For every task chain other than the CUA, the interference is bounded by its upper arrival curve.*

$$\forall b \neq a, \tau_i \in \mathcal{T}_b : \vartheta^{(1)}_{a,i}(q) = \max(\eta^+_b(B_a(q)), 1) \tag{10}$$

Proof. The arrival curve is an upper bound for the number of activations of $\mathcal{T}_b$ that arrive within the busy window. Due to Definition 4.8, the busy window starts with no pending activations of $\tau_i \in \mathcal{T}_b$ and any additional interferers or blockers that delay the execution of τ_i must also be included in the busy window. Hence, any situation that may lead to a backlog worse than the arrival curve is part of the busy window and there are no pending activations of τ_i at the end of the busy window. □

By definition of the q-event busy window, we can further limit the activations of the last task of the CUA and any of its strict predecessors.

Corollary 4.15. *The last task τ_i of the CUA ($\mathcal{T}_a$) never executes more than q times.*

$$\vartheta^{(2)}_{a,i}(q) = q \tag{11}$$

The same holds for the strict predecessors of τ_i in the CUA.

$$\forall \tau_j \in strpred^*_i \cap \mathcal{T}_a : \vartheta^{(2)}_{a,j}(q) = q \tag{12}$$

Let us now focus on the precedence relations. The authors of [19] have shown that every interfering task $\tau_i \notin \mathcal{T}_a$ can execute at most once if it has a lower-priority predecessor τ_j which can never execute during the busy window (cf. H_d in Figure 6). As this bases on the fact that two executions of τ_i are always separated by an execution of τ_j, we can also apply this in case of strict precedence between those tasks in which case it is applicable in both directions. Note that this changes τ_i into a deferred interferer so that it may leave pending activation after the busy window (cf. H_d in Figure 6). Nonetheless, this bound will still hold independent of any pending or arriving activations.

Corollary 4.16. *A task can execute at most once if there is a lower-priority or strict predecessor that cannot execute at all within the busy window of the CUA.*

$$\forall \tau_i \notin \mathcal{S}i, \tau_j \in \mathcal{S}i : \vartheta^{(3)}_{a,i}(q) = \begin{cases} 1 & if \quad n_{a,j}(q) = 0 \\ \infty & else \end{cases} \tag{13}$$

with $\mathcal{S}_i = \{\tau_j | \tau_j \in \mathcal{L}_i \cap pred^*_i \vee \tau_j \in strpred^*_i \cup strsucc^*_i\}$.

The formulated bounds for the precedence relations come into effect when we consider the priorities and exclude lower-priority tasks from the busy window that can never act as blockers.

More precisely, blocking occurs when there is a task τ_i that shares an execution context with the CUA such that, in the worst case, the latter must wait until the execution context is released. Note that we also need to consider indirect (transitive) blocking (cf. B_i in Figure 6), which has most recently been described in [6]. Transitive blocking occurs when there is a task τ_j which blocks another execution context from τ_i and which is no strict predecessor or successor of the latter. This is due to the fact that a strict predecessor (successor) can only be an indirect blocker if it does not block the CUA directly, which means that any execution context shared with the CUA is allocated after (released before) it.

We therefore define the set of tasks that are potentially required to execute within the busy-window of $\mathcal{T}_a$ although they run on a lower priority. We do this conservatively by assuming that every higher-priority task may be considered as arriving interferer. Hence, we include every lower-priority task if it may block the CUA, a higher-priority task, or another lower-priority blocker. Note that blocking of higher-priority tasks can only occur if there is no precedence relation (directly or indirectly) between the tasks. Otherwise, the higher-priority task is deferred and we do not need to consider the blocking (cf. Corollary 4.16).

Definition 4.17. The possible lower-priority blockers are found among all tasks that block an execution context which is also blocked by the CUA, or by another possible blocker or higher-priority task that is not a predecessor or successor.

$$\begin{aligned}\mathcal{B}_a = \{\tau_j \notin \mathcal{H}_a^c \cup \mathcal{T}_a | V_e(\tau_j) \cap V_e(\mathcal{T}_a) \neq \emptyset \\ \vee \exists \tau_i \in \mathcal{B}_a \cup \mathcal{H}_a^c : V(\tau_i) \cap V(\tau_j) \neq \emptyset \wedge \tau_j \notin pred_i^* \cup succ_i^*\}\end{aligned} \tag{14}$$

with $V_e(X) = \{e \in V_e | \exists \tau_i \in X \wedge \{\tau_i, e\} \in E_e\}$.

In the scope of this paper, we do not further restrict the interference from these lower-priority blockers, e.g., by considering mutually exclusive blockers or single-time blocking. Although this can be pessimistic in certain scenarios, our evaluation did not show overly pessimistic results (cf. Section 6). Instead, we thereby avoid several pitfalls that arise from the fact that we can construct scenarios in which the CUA might be blocked multiple times by the same lower-priority blocker (cf. H_{b2} in Figure 6).

The remaining, i.e., non-blocking, lower-priority tasks can now be excluded if they do not have a lower-priority task that might execute during the busy window of $\mathcal{T}_a$.

Theorem 4.18. *For static-priority scheduling, all lower-priority tasks τ_i that are neither part of the CUA nor a possible lower-priority blocker will never be scheduled during the busy window of the CUA unless there is an even lower-priority task τ_j that may execute in this busy window.*

$$\forall \tau_i \notin (\mathcal{H}_a^c \cup \mathcal{T}_a \cup \mathcal{B}_a) :$$

$$\vartheta_{a,i}^{(4)}(q) = \begin{cases} 0 & if\ \mathcal{L}_i = \emptyset \\ 0 & if\ \nexists \tau_j \in \mathcal{L}_i : n_{a,i}(q) > 0 \\ \infty & else \end{cases} \tag{15}$$

Proof. At first, the bound is not applied for tasks that are part of the CUA, a higher-priority interferer or a (direct or indirect) lower-priority blocker. Hence, the remaining tasks τ_i can only execute within the busy window of the CUA if they interfere with any other task in the busy window. As we excluded blocking interference, this can only happen if there is a task on a lower-priority than τ_i whose execution cannot be safely excluded from the busy window. □

Note that Equation (15) can be improved (and simplified) if a donation/helping mechanism as in [23] is applied such that a lower-priority blocker will never see any lower-priority interference if a higher-priority task is waiting for it.

5 RELATED WORK

There exists a large body on research regarding timing/response-time analysis in general. In this section, we provide a brief overview of related work in three different subdomains: modelling for timing analysis, timing analysis considering precedence relations, and timing analysis considering blocking/shared resources.

Stigge [24] intensively studied schedulability analysis for various (graph-based) task models of different expressiveness. More specifically, he proposed the Digraph Real-Time task model which already increases the expressiveness of existing models but does not consider blocking relations due to locking, semaphores or shared resources. MAST [1] is based on a model which originated from the very expressive but also complex MARTE UML [3] and focuses on the analysis of object-oriented real-time systems. This model is quite similar to our approach and particularly includes shared resources but restricts these to be unlocked within the same segment, i.e., critical sections must not cross scheduling servers.

MAST implements various offset-based analyses, such as [13], which exploit precedence relations to improve the RTA by formulating correlations between activations (inputs) that would otherwise be assumed independent/uncorrelated. For instance, such correlations have also been formulated between parallel paths with a common timing reference [17] and for the sequential character of output events from a single scheduler [10, 18]. All these can be considered orthogonal to our approach as they focus on reintroducing correlations that are hidden in the scope of a local timing analysis. Kurtin et al. [11] exploited precedence relations in streaming applications by augmenting the RTA with dataflow analysis and thereby cover cyclic dependencies but no blocking. We believe that this approach can be combined with our methods. Schlatow and Ernst [19] applied the busy-window concept to entire task chains and showed a substantial improvement over a compositional performance analysis (CPA), which computes the chain's response time by summing up the response times of all tasks separately. However, this work only allows strictly sequential chains (no forks/joins) and strictly synchronous/asynchronous communication. Nevertheless, this work demonstrated that the task-chain busy-window approach substantially increases the tightness and scalability of the CPA. Schliecker and Ernst [20] presented a path analysis that improves the latency bounds for global task chains by addressing their self-interference in case of pipelined behaviour. However, this still relies on performing a compositional performance analysis in advance, which tends to be overly pessimistic or not convergent on a single processing resource with dependent tasks.

When it comes to timing analysis under blocking effects, the large body of research considering shared resources in multiprocessor systems appears to be related to our work. However, our model addresses blocking relations due to local resource sharing, which is typically not considered. Negrean and Ernst [14] considered a nested locking of global and local resources that induced additional blocking on local tasks. The crucial effects of nested locks and the resulting transitive blocking have most recently been studied by Biondi et al. [6]. Nevertheless, to the best of our knowledge the existing literature always assumes that shared resources are released upon task/job completion (or segments in case of MAST), which is a reasonable assumption in the context of shared resources in multiprocessors systems but only of limited applicability in our scenario.

6 EVALUATION

For the evaluation, we implemented our analysis in Python as an extension to pyCPA [4] based on a constraint-programming approach in order to address the propagation of the $n_{a,i}$ values. We formulate our models as GraphML descriptions that can be fed into this analysis extension. Note that our pragmatic implementation has not been optimised for performance.

Table 1. Worst-case Response-Time Results Chains A, B and C (As Defined in Section 3.3) for All Priority Assignments in Our Illustrative Use Case (CPA Results in Parentheses).

	Priority of ctx_a/ctx_b/ctx_c ($H = High, M = Medium, L = Low$)					
	H/M/L	H/L/M	M/H/L	M/L/H	L/H/M	L/M/H
C	70 (90)	70 (90)	70 (180)	70 (180)	90 (270)	90 (270)
A	70 (240)	90 (330)	70 (210)	90 (390)	90 (270)	90 (360)
B	90 (330)	70 (240)	90 (390)	70 (210)	90 (360)	90 (270)

Unfortunately, we cannot refer to any reference benchmarks due to the novelty of our model. Thus, we first have a look at our use case introduced in Section 3.1, to show how our analysis covers the targeted scenarios. In addition, we compare our approach with a use case from the related work and run some synthetic but characteristic benchmarks to test the scalability.

6.1 Illustrative Use Case

We first performed our RTA on the illustrative use case shown in Figure 5. For this, we set the best-case (worst-case) execution times of each task to 5 (10) time units and chose a long activation period of 1000 time units to model a sporadic activation without bursts.

In particular, we performed the RTA for single tasks and the full chains (cf. Section 3.3), and compared both with the CPA results from the unmodified pyCPA for all six priority permutations. For the single-task analysis, both the CPA and our approach provide the same results for most tasks. However, for the tasks allocating P the CPA provides either optimistic results because the blocking is not accounted or overly pessimistic results if we add the longest response-time of any lower-priority blocker.

The results for the full-chain analysis are shown in Table 1. On the one hand, although omitting the blocking effects, the CPA results are overly pessimistic due to the fact that it sums up the worst-case response times of all tasks in a chain. On the other hand, our results are not tight either as we can see from the results for chain C in the first priority assignment. This chain has three tasks ($\tau 11$, $\tau 12$, $\tau 13$) and therefore a (worst-case) core execution time of 30. Our analysis accounted for a single interference from tasks $\tau 21$, $\tau 22$, $\tau 23$ and $\tau 32$ because the latter was considered as a lower-priority blocker and the others as interferers of this blocker. In the general case, it is correct to account for the blocking from $\tau 32$ despite the precedence relations between the chains since the blocking might originate from a previous activation. However, in this particular scenario, $\tau 32$ will also act as a blocker for $\tau 22$ so that their execution can be considered mutually exclusive within the busy window of C. Note that we have not included this aspect in the scope of this paper, but it can be added to our analysis approach in terms of a candidate search. As mentioned in Section 3.2, despite all similarities, MAST is not applicable to this use case.

6.2 Comparison with Related Work

In this section, we compare our analysis with related work on the analysis of task-chain response times. We therefore model the park assist use case introduced in [19]. This use case comprises seven software components that build two independent task chains that represent two different functional chains: a park assist and lane assist function. Note that this use case only covers strict precedence relations and excludes blocking relations between task chains.

Figure 7 depicts our model of this unmodified use case. The related work provides a basic and a refined analysis for this scenario. Using the same best-case/worst-case execution times, we get the same task-chain response-time results as the basic analysis for all 5040 possible priority assignments whereas the refined analysis still provides better results for chain P in 1332 assignments and

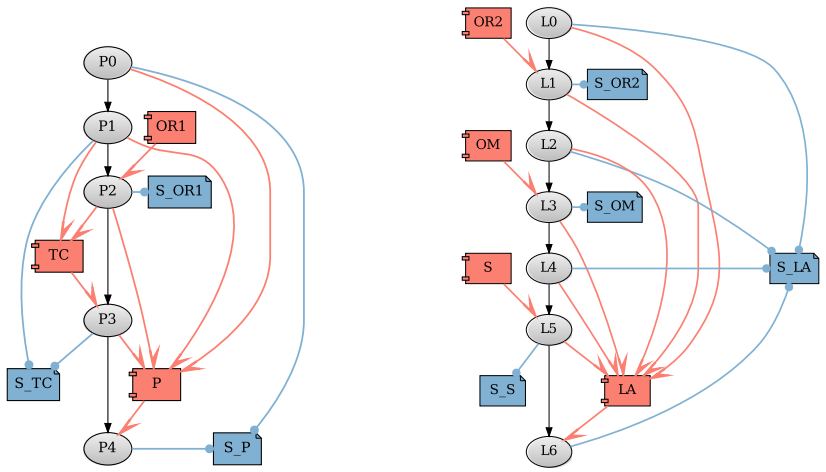

Fig. 7. Model representation of the park assist (tasks P0-P4) and lane assist (tasks L0-L6) use case in [19].

Table 2. Worst-Case Response-Time Results and Core Execution Times (CET) for the Modified Park Assist Use Case with Blocking and Priority Inheritance (MAST Results in Parentheses)

	H/L	L/H	CET
P	36 (45)	76 (85)	26
L	76 (85)	60 (75)	50

for chain L in 672 assignments because it considers mutual exclusion between different segments of the interfering chains. As mentioned before, this can only be covered by our approach if we combine it with a candidate search.

This use case only becomes more interesting for our approach if we add blocking between the two, otherwise independent, task chains. We easily achieve this by letting L1 and P2 share the same execution context (OR2). By also assuming priority inheritance for the IPC, we render this use case analysable with MAST. The latter results in all tasks of each chain to be mapped to the same scheduling context. The resulting model thus comprises only two scheduling contexts and therefore only two different priority assignments. The results of our analysis are shown in Table 2, which also shows the core execution time of each chain and the results from an offset-based analysis with MAST (in parentheses). This illustrates that the higher priority chain experiences an additional blocking delay equal to the worst-case execution time of L1/P2 respectively whereas MAST's results are more pessimistic.

We also tried to analyse this use case with the unmodified pyCPA. While we expected optimistic results because it does not consider blocking, even in this simple scenario, it was not able to converge and return any results although the processor load was only at 65%.

6.3 Synthetic Benchmarks

As previously mentioned, we are not aware of any reference benchmarks which comprise the kind of precedence and blocking relations that we address with this work. Although there exist benchmark suites covering quite similar scenarios, such as E3S[1], they do not describe the

[1] http://ziyang.eecs.umich.edu/dickrp/e3s/.

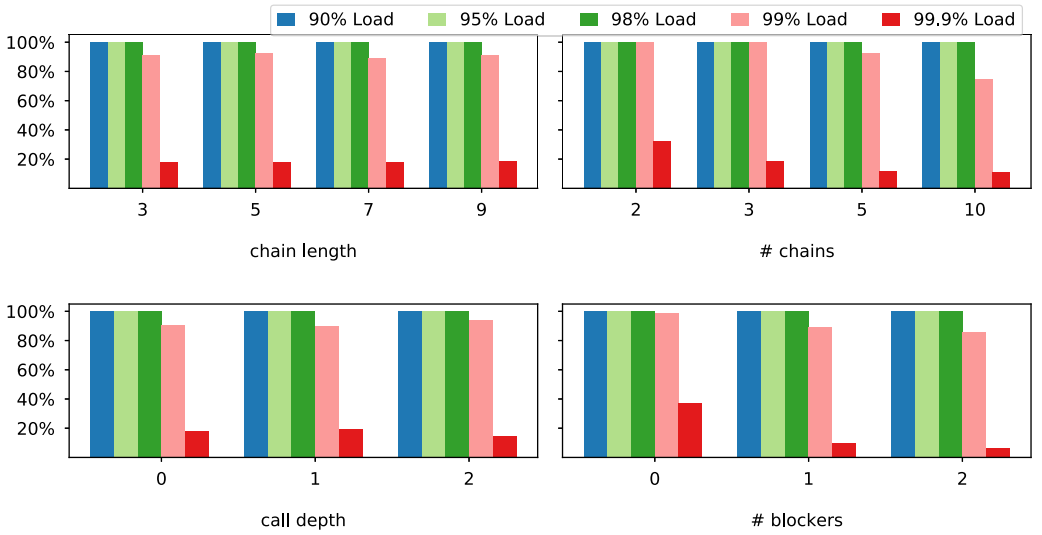

Fig. 8. Percentage of analysable models for varying system parameters and processor load.

software system to the level of detail at which we are aiming. In order to evaluate our analysis in a wider range of scenarios, we tried to parametrise our target systems such that we can randomly generate models with different characteristics. As our main parameters, we chose chain length, number of chains, maximum number of blockers, and call depth, which is the maximum number of nested allocations of execution contexts. Note that we restricted the models to sequential task chains without branches as these had put to many restrictions on the combination of the other parameters. For the resulting task chains, we choose a periodic activation pattern with a jitter of 10%. For each model we randomly generated worst-case execution times to generate different levels of processor utilisation/load up to almost 100%. We randomly selected different priority assignments and performed our RTA. As we did not impose any deadline constraints, we only test the convergence and scalability of the analysis. More precisely, our assumption is that a perfectly tight and scalable analysis will be able to calculate response times even for a fully loaded system. However, as soon as over approximation comes into account, the analysis will not be able to converge (i.e., find a Q_a) in a reasonable number of iterations.

The results are depicted in Figure 8, which shows the percentage of priority assignments that could be successfully analysed. In the remaining cases, the busy-window grew too large (we aborted the analysis at $Q_a = 100$). We observe that most of the systems were considered schedulable up to 98% load, which indicates that our analysis cannot be overly pessimistic. Moreover, we notice that an increasing number of chains and an increasing number of blockers has a significant impact on the analysability for the systems with 99% and 99.9% load. On the other hand, there is no such correlation for the chain length or the call depth.

7 CONCLUSION

In this paper, we addressed the RTA for software systems that build upon communicating threads or software components such as microkernel-based systems. When modelling these systems for timing analysis, those systems not only result in tasks with precedence relations but also with mutual blocking effects if the software components can only be re-entered at certain points in their execution. For these scenarios, we presented a novel task model that explicitly describes the

precedence relations and the mutual blocking between tasks as well as where scheduling decisions are made by the operating system (OS). Given sufficient knowledge about the OS, we can easily translate sequence diagrams – which naturally describe inter-component communication – into platform-specific task graphs in order to perform a RTA. For this RTA, we proposed a framework that is applied as a single-processor analysis in the scope of a CPA. This framework deals with an arbitrary number of independent workload or event-count bounds on the interference that can be experienced by the chain under analysis. In the scope of this paper, we focused on formulating conservative event-count bounds for static-priority scheduling that cover all the tricky aspects such as priority inversion and deferred activations. This represents a significant improvement over state-of-the-art techniques as these aspects cannot be neglected in realistic systems in general, especially because they may lead to hard-to-catch timing errors. Our experimental evaluation additionally showed that our analysis can already provide reasonably tight results although our event-count bounds include pessimistic assumptions.

ACKNOWLEDGMENTS

This work was funded within the DFG Research Unit *Controlling Concurrent Change*, funding number FOR 1800. The authors particularly want to thank Mischa Möstl for the fruitful discussions regarding our modelling approach.

REFERENCES

[1] 2000-2015. MAST: Modeling and Analysis Suite for Real-Time. (2000-2015). Retrieved 2017-04-04 from http://mast.unican.es/.

[2] 2001-2017. QNX Neutrino RTOS. (2001-2017). http://www.qnx.com/products/neutrino-rtos/neutrino-rtos.html.

[3] 2009-2011. MARTE UML: Modeling and Analysis of Real-Time Embedded Systems. (2009-2011). Retrieved 2017-04-04 from http://www.omg.org/spec/MARTE/.

[4] 2010-2017. pyCPA website and source code. (2010-2017). Retrieved 2017-04-04 from https://bitbucket.org/pycpa.

[5] 2015. MAST 1.5.0: Description of the MAST Model. (2015). Retrieved 2017-04-04 from http://mast.unican.es/mast_description.pdf.

[6] Alessandro Biondi, Björn B. Brandenburg, and Alexander Wieder. 2016. A Blocking Bound for Nested FIFO Spin Locks. In *Real-Time Systems Symposium (RTSS)*. IEEE, 291–302.

[7] Kevin Elphinstone and Gernot Heiser. 2013. From L3 to seL4 – What Have We Learnt in 20 Years of L4 Microkernels? In *ACM Symposium on Operating Systems Principles*. Farmington, PA, USA, 133–150.

[8] Norman Feske. 2017. *Genode OS Framework Foundations 17.05*. Technical Report.

[9] Rafik Henia, Laurent Rioux, Nicolas Sordon, Gérald-Emmanuel Garcia, and Marco Panunzio. 2015. Integrating Formal Timing Analysis in the Real-Time Software Development Process. In *Workshop on Challenges in Performance Methods for Software Development (WOSP'15)*. ACM, New York, NY, USA, 35–40.

[10] Steffen Kollmann, Victor Pollex, and Frank Slomka. 2011. Reducing Response Times by Competition Based Dependencies. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV), Oldenburg, Germany, February 21-23, 2011*. 91–100.

[11] P. S. Kurtin, J. P. H. M. Hausmans, and M. J. G. Bekooij. 2016. Combining Offsets with Precedence Constraints to Improve Temporal Analysis of Cyclic Real-Time Streaming Applications. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 1–12.

[12] Jochen Liedtke. 1993. Improving IPC by kernel design. *ACM SIGOPS Operating Systems Review* 27, 5 (Dec. 1993), 175–188.

[13] Jukka Mäki-Turja and Mikael Nolin. 2008. Efficient implementation of tight response-times for tasks with offsets. *Real-Time Systems* 40, 1 (2008), 77–116.

[14] Mircea Negrean and Rolf Ernst. 2012. Response-Time Analysis for Non-Preemptive Scheduling in Multi-Core Systems with Shared Resources. In *Symposium on Industrial Embedded Systems (SIES)*. Karlsruhe, Germany.

[15] Gabriel Parmer. 2010. The Case for Thread Migration: Predictable IPC in a Customizable and Reliable OS. In *Intern. Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*. Brussels, Belgium.

[16] Simon Perathoner, Tobias Rein, Lothar Thiele, Kai Lampka, and Jonas Rox. 2010. Modeling Structured Event Streams in System Level Performance Analysis. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*. ACM, Sweden, 37–46.

[17] Rolf Ernst Rafik Henia. 2006. Improved Offset-Analysis Using Multiple Timing-References. In *Design, Automation and Test in Europe (DATE)*.

[18] Jonas Rox and Rolf Ernst. 2010. Exploiting Inter-Event Stream Correlations Between Output Event Streams of non-Preemptively Scheduled Tasks. In *Design, Automation and Test in Europe (DATE)*.

[19] Johannes Schlatow and Rolf Ernst. 2016. Response-Time Analysis for Task Chains in Communicating Threads. In *Real-Time Embedded Technology and Applications Symposium (RTAS)*. Vienna, Austria.

[20] Simon Schliecker and Rolf Ernst. 2009. A Recursive Approach to End-To-End Path Latency Computation in Heterogeneous Multiprocessor Systems. In *Intern. Conf. on Hardware Software Codesign and System Synthesis (CODES-ISSS)*. ACM, Grenoble, France.

[21] Simon Schliecker, Jonas Rox, Matthias Ivers, and Rolf Ernst. 2008. Providing Accurate Event Models for the Analysis of Heterogeneous Multiprocessor Systems. In *6th Intern. Conf. on Hardware Software Codesign and System Synthesis (CODES-ISSS)*. Atlanta, GA.

[22] L. Sha, R. Rajkumar, and J. P. Lehoczky. 1990. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Trans. Comput.* 39, 9 (Sept. 1990), 1175–1185.

[23] Udo Steinberg, Alexander Böttcher, and Bernhard Kauer. 2010. Timeslice Donation in Component-Based Systems. In *Intern. Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*. Brussels, Belgium.

[24] Martin Stigge. 2014. *Real-time workload models: Expressiveness vs. analysis efficiency*. Ph.D. Dissertation. Uppsala University.

[25] Sebastian Tobuschat, Rolf Ernst, Arne Hamann, and Dirk Ziegenbein. 2016. System-level Timing Feasibility Test for Cyber-physical Automotive Systems. In *Symposium on Industrial Embedded Systems (SIES)*.

Received March 2017; revised June 2017; accepted June 2017