A Real-time Calculus Approach for Integrating Sporadic Events in Time-triggered Systems

Anaïs Finzi
TTTech Computertechnik AG, Austria

Silviu S. Craciunas
TTTech Computertechnik AG, Austria

Marc Boyer
ONERA / DTIS, Universitè de Toulouse, France

arXiv:2204.10264v1 [cs.NI] 21 Apr 2022

Abstract

In time-triggered systems, where the schedule table is predefined and statically configured at design time, sporadic event-triggered (ET) tasks are handled within specially dedicated slots or when time-triggered (TT) tasks finish their execution early. We introduce a new paradigm for synthesizing TT schedules that guarantee the correct temporal behavior of TT tasks and the schedulability of sporadic ET tasks with arbitrary deadlines. The approach first expresses a constraint for the TT task schedule in the form of a maximal affine envelope that guarantees that as long as the schedule generation respects this envelope, all sporadic ET tasks meet their deadline. The second step consists of modeling this envelope as a burst limiting constraint and building the TT schedule via simulating a modified Least-Laxity-First (LLF) scheduler. Using this novel technique, we show that we achieve equal or better schedulability and a faster schedule generation for most use-cases compared to other approaches inspired by, e.g., hierarchical scheduling. Moreover, we present an extension to our method that finds the most favourable schedule for TT tasks with respect to ET schedulability, thus increasing the probability of the computed TT schedule remaining feasible when ET tasks are later added or changed.

2012 ACM Subject Classification Computer systems organization → Dependable and fault-tolerant systems and networks

Keywords and phrases time-triggered, event-triggered, scheduling, real-time, real time calculus

1 Introduction

Time-triggered systems have been used to a great extent in the aerospace domain where the safety-critical nature of the applications imposes a certain level of determinism on the architecture, especially when certification is required [23, 15]. Moreover, the automotive sector has recently seen a push towards centralizing functionality onto a more scalable and flexible integrated platform (c.f. [52]) in order to support the complex real-time needs of, e.g., ADAS subsystems [24, 47] and to allow a mixed-criticality paradigm. Thus, the use of time-triggered scheduling (cyclic executive) solutions leading to more deterministic (and thus more easily verifiable and certifiable) systems is also gaining importance in the automotive domain [45, 59, 20]. In particular, the complex jitter and multi-rate cause-effect requirements found in ADAS applications [5, 6] cannot be easily guaranteed off-line using classical fixed- or dynamic-priority approaches and necessitate a more predictable time-triggered architecture (TTA) [47]. While TTA has many benefits in terms of predictability, stability, compositionality, and determinism, the use of a static schedule table is notoriously inefficient at integrating sporadic event-driven tasks (ET). Conversely, pure event-triggered systems suffer from many drawbacks compared to a time-triggered execution, e.g., high jitter and starvation (c.f [56, 74, 44, 30, 34]). Modern safety-critical systems benefit most from combining the two paradigms, allowing a time-triggered system to be flexible enough to respond to sporadic events when needed.

For time-triggered systems, where the schedule table is predefined and statically configured at design time, sporadic event-triggered (ET) tasks are handled within specially allocated slots or when time-triggered (TT) tasks finish their execution earlier than their worst-case assumption. While there is a significant body of work (c.f.[48] for an extensive survey) concerning pure time-triggered schedule generation, which is an NP-complete problem, most of the methods do not consider the schedulability of sporadic ET tasks. Traditionally, the integration of sporadic ET tasks in time-triggered systems is either done via a feedback loop integrated into the TT schedule generation mechanism [54, 55], or via hierarchical scheduling [1, 61, 62, 58]. For both approaches, the computational effort (on top of the complexity of creating TT schedules) can be significant due to the response time analysis for each variation of TT slot placement or due to solving the server design problem within the TT schedulability space. Therefore, the challenge is to create static schedule tables for which both TT and ET tasks respect their deadlines while keeping the computational effort low.

We present a novel approach in which we first compute a maximal affine envelope (defined by a maximum burst and a rate) for the TT tasks in the system, such that as long as a TT schedule respects this envelope, all sporadic ET tasks meet their deadline. The second step involves expressing this envelope as a burst limiting constraint on the TT schedule and building the static schedule table via simulating a modified Least-Laxity-First (LLF) scheduler. Using this novel technique, we trade-off complexity for exactness via the pessimism of the affine envelope approximation resulting in a faster schedule generation while still achieving equal or better schedulability compared to more runtime-intensive approaches. Moreover, this method enables an efficient design optimization technique for iterative design processes where ET tasks are added or changed later. Our contributions are, therefore:

- a new and efficient approach based on affine envelope approximations for guaranteeing the schedulability of ET tasks with arbitrary deadlines without the need for complex response-time analysis,
- a novel LLF-based algorithm that respects the affine envelope (expressed as a burst limiting constraint) and produces a static schedule guaranteeing both TT and ET deadlines,
- a computationally "cheap" method for integrating ET tasks in TT systems that, while being pessimistic for some task sets, has in most cases equal or better schedulability and runtime results compared to other methods based on the traditional hierarchical response-time-based or polling approaches,
- a design optimization where we maximize the solution space for changing or adding ET tasks without modifying the existing TT schedule generated via our method.

We start by introducing some necessary preliminaries in Section 2, followed by a review of related literature in Section 3. We introduce two polling approaches that follow earlier results in Section 4 and our novel method based on affine envelope approximations in Section 5. After evaluating our method in Section 6 we draw some conclusions in Section 7.

2 Preliminaries

2.1 System model

We assume a task dispatcher that schedules TT tasks based on an offline generated static schedule table (cyclic executive), and in the slots that are left for ET tasks, implements a 2nd-level preemptive scheduler based on fixed priorities (c.f. Figure 1). We denote the set of TT and ET tasks with $\mathcal{T}^{TT}$ and $\mathcal{T}^{ET}$, respectively. A TT or ET task τ_i is defined by the tuple (p_i, C_i, T_i, D_i) with C_i denoting the computation time, p_i is the task priority, and D_i

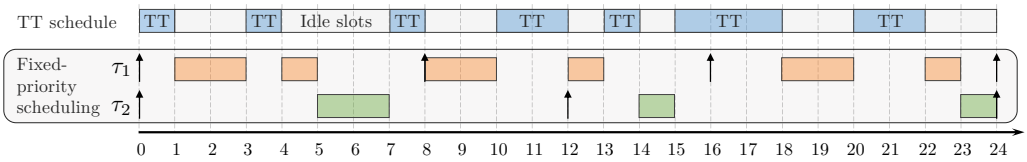


Figure 1 Static schedule table with 2nd-level fixed-priority scheduling in idle slots.

the relative deadline of the task. For TT tasks, T_i represents the period, while for ET tasks where we assume a sporadic model, it describes the minimal inter-arrival distance (MIT). Usually, TT tasks have a constrained-deadline model ($T_i \leqslant D_i$), while ET tasks can have an arbitrary deadline, i.e., it can also be larger than the inter-arrival time. For convenience, we say that all TT tasks share the same (highest) priority. Event-triggered tasks, having a lower priority than TT tasks, are indexed in the order of their relative priority, i.e., τ_i has higher priority than τ_j ($p_i < p_j$) implies $i < j$, but several tasks may have the same priority ($p_i = p_j$) in which case they are selected in FIFO order.

The timeline of scheduling decisions is divided into equal segments by the microtick mt, representing the smallest scheduling granularity for tasks [35, 34]. Usually, the granularity of the timeline is in the range of hundreds of microseconds to a few milliseconds; however, we do not assume any lower value here as the granularity in, e.g., embedded devices with custom runtime systems can go down to the order of microseconds (e.g. [14]). In the following, we assume that any D, C, T are multiples of mt. A schedule for a finite set of tasks $\mathcal{T} = \mathcal{T}^{TT} \cup \mathcal{T}^{ET}$ is a partial function $\sigma : \mathbb{N} \hookrightarrow \mathcal{T}$ from the time domain to the set of tasks, that assigns to each interval $[t \cdot mt, (t+1) \cdot mt)$ defined by the microtick granularity a task that is running in that time interval. We assume that each schedule σ repeats after a certain time period called the schedule cycle, which is usually equal to the hyperperiod of the system, defined by $T = lcm_{\tau_i \in \mathcal{T}^{TT}} \{T_i\}$. Furthermore, we assume that the system is not overloaded, *i.e.* $U \leqslant \lambda$, with λ the computation capacity of the system and hence $\sigma(t)$ is uniquely defined for each point on the microtick timeline. We consider in this work that the tasks from both sets $\mathcal{T}^{TT}$ and $\mathcal{T}^{ET}$ are scheduled on a single-core CPU with capacity $\lambda = 1$.

We introduce a few notations to ease readability. For any task τ_i, $p(i)$ is the priority of the task, $U_i = \frac{C_i}{T_i}$ is the utilization of the task τ_i, $U^{TT} = \sum_{\tau_i \in \mathcal{T}^{TT}} U_i$ is the utilization of all TT tasks, $U^{ET} = \sum_{\tau_i \in \mathcal{T}^{ET}} U_i$ is the utilization of all ET tasks, $U_{>p} = \sum_{\tau_i \in \mathcal{T}, p(i) > p} U_i$ is the utilization of all tasks with priority higher than p, $U^{ET}_{=p} = \sum_{\tau_i \in \mathcal{T}^{ET}, p(i) = p} U_i$ is the utilization of all ET tasks with priority equal to p, $U^{ET}_{>p} = \sum_{\tau_i \in \mathcal{T}^{ET}, p(i) > p} U_i$ is the utilization of all ET tasks with priority higher than p, and notice that if p is the priority of an ET task, $U_{>p} = U^{TT} + U^{ET}_{>p}$ since the TT task have higher priority than ET tasks. Using the same pattern, we define $C^{TT}, C^{ET}_{>p}, C^{ET}_{=p}, C^{ET}_{\geqslant p}$ and notice that $C^{ET}_{>p} + C^{ET}_{=p} = C^{ET}_{\geqslant p}$.

2.2 Real-time (and network) calculus

Network Calculus (NC) [38] is a theory for quantifying worst-case (latency and backlog) bounds in computer networks, using min-plus and max-plus algebra to relate the minimum service of network nodes and the maximum amount of flow traffic. Real-time calculus (RTC) [71] is the real-time systems equivalent of NC for modeling and analyzing the worst-case behavior of tasks (e.g. [68]). It also uses non-decreasing functions to model the maximum task computation demand and the minimum available CPU computation service in any specified time interval. It can thus be seen as a variant of the classical NC framework with some minor differences [69]. We only introduce the most important definitions and refer the

reader to [71, 49] for a more in-depth description.

The response time (i.e., delay) of a task τ_i of a set of tasks $\mathcal{T}$ is detailed in Theorem 3. It depends on i) the maximum amount of requested computation of $\mathcal{T}$, represented by a so-called arrival curve $\alpha(t)$ defined in Definition 1, and ii) the minimum computation capacity available to $\mathcal{T}$, represented by a so-called minimum service curve $\beta(t)$ defined in Definition 2. Additionally, we define γ in the maximum service curve, which represents the maximum amount of computation available to $\mathcal{T}$. As introduced in [71], the request function $R(t) \geqslant 0$ constitutes the accrued computation time solicited until time t. Conversely, the function $\mathcal{C}(t) \geqslant 0$ is the maximum computation time delivered until time t [71].

▶ **Definition 1** (Arrival curve [71]). *The arrival curve $\alpha(t)$ of a request function $R(t)$ is a non-decreasing function which satisfies:*

$$R(t) - R(s) \leqslant \alpha(t-s), \forall s \leqslant t. \tag{1}$$

▶ **Definition 2** (Service curves [71, 12, 69]). *The maximum service curve $\gamma(t)$ and minimum service curve $\beta(t)$ of a capacity function $\mathcal{C}(t)$ are non-negative and non-decreasing functions satisfying:*

$$\beta(t-s) \leqslant \mathcal{C}(t) - \mathcal{C}(s) \leqslant \gamma(t-s), \forall s \leqslant t. \tag{2}$$

We now reiterate the main result for computing bounds on delay.

▶ **Theorem 3** (Maximum response time [12]). *For a task dispatcher offering a minimum service curve $\beta(t)$ to a set of tasks $\mathcal{T}$ with an arrival curve $\alpha(t)$, the worst-case response time of a task is the maximum horizontal distance $hDev(\alpha, \beta)$ computed between $\alpha(t)$ and $\beta(t)$.*

To compute the response time of any priority, we use the service curve in Theorem 4.

▶ **Theorem 4** (Minimum remaining service curve [73]). *For a preemptive fixed-priority dispatcher of computation capacity λ, and a set of tasks $\tau_i \in \mathcal{T}$ with priorities $p(i)$, the minimum service curve remaining to tasks of priority p is the non-decreasing positive function*

$$\beta_p^{SP}(t) = [\lambda \cdot t - \alpha_{>p}(t)]_{\uparrow}^{+} \text{ , with } \alpha_{>p}(t) = \sum_{\tau_i \in \mathcal{T}, p(i) > p} \alpha_i(t). \tag{3}$$

Similar to the NC considerations for sporadic flows and rate-latency servers [10], if a task generates jobs of cost $C \in \mathbb{R}^+$ at a rate given by the period (or minimal inter-arrival time) $T \in \mathbb{R}^+$, it admits the linear arrival curve $\alpha_{r,b} : \mathbb{R}^+ \to \mathbb{R}^+$, $0 \mapsto 0$ and $t \mapsto rt + b$ if $t > 0$, with $r = \frac{C}{T}$ and $b = r \cdot T$. When tasks are scheduled by a dispatcher in a certain slot, they are usually executed a constant rate R (using one unit of computation for every unit of time) after some delay (latency) L which is due to blocking by e.g. other higher-priority tasks. This matches a rate-latency service [10], modelled by a function $\beta_{R,L} : t \mapsto R \cdot [t - L]^+$.

▶ **Corollary 5** (Linear maximum response time). *Consider a linear arrival curve $\alpha_{r,b}(t)$ and a rate-latency service curve $\beta_{R,L}(t) = R \cdot [t - L]^+$, then $hDev(\alpha_{r,b}, \beta_{R,L}) = L + \frac{b}{R}$.*

▶ **Proposition 1.** *Let $r, r', b, b', R, L \in \mathbb{R}^+$ be some parameters. Then, we have $\alpha_{r,b}(t) + \alpha_{r',b'}(t) = \alpha_{r+r',b+b'}(t)$. If $r \leqslant R$, then, $hDev(\alpha_{r,b}, \beta_{R,L}) = L + \frac{b}{R}$ and $[\beta_{R,L}(t) - \alpha_{r,b}(t)]_{\uparrow}^{+} = \beta_{(R-r), \frac{RL+b}{R-r}}(t)$.*

The proofs can be found in [9, Prop. 3.7].

3 Related work

Sporadic events can be readily integrated alongside periodic real-time tasks in the schedulability analysis of both fixed- and dynamic-priority systems using, e.g., Deadline Monotonic (DM) or Earliest-Deadline First (EDF) schedulability tests (c.f [43, 3, 4, 7, 75]). However, it may be helpful to have some form of isolation in the temporal domain between certain task types, e.g., between periodic/sporadic and aperiodic tasks. The temporal isolation is usually achieved via bandwidth servers, where a bandwidth server is defined as a periodic task with a budget and period that can handle one or more aperiodic events. The bandwidth server can then be scheduled alongside the periodic or sporadic tasks via, e.g., fixed-priority or dynamic-priority dispatchers. One of the main goals of bandwidth servers (beyond temporal isolation) is to minimize the response time of aperiodic tasks, but no guarantees such as deadlines can be given for aperiodic events. Examples of bandwidth servers for aperiodic ET task handling are the Polling Server [40, 65], the Deferrable Server [67], or the Sporadic Server in both the FP [26, 65] and EDF [66] variants.

In systems with a time-triggered scheduler, integrating sporadic event-based tasks is more challenging than in purely fixed- or dynamic-priority systems. In [17, 53], the authors present a "holistic" schedulability analysis and design optimization approach for systems where tasks are event-triggered, but the communication backbone is based on the time-triggered bus protocol TTP. In [54, 55], a mixed time- and event-triggered application model similar to ours is considered for distributed embedded systems. The authors first present an analysis of ET task schedulability given a pre-defined TT schedule and then use a list-scheduling-based heuristic with limited backtracking to guide the generation of TT task schedules also to increase ET task schedulability. The approach in [54, 55] favors the correctness of TT schedules over ET schedulability and is not guaranteed to find a feasible schedule for both TT and ET tasks. In that respect, it is more similar to a greedy method for generating TT schedules, checking the schedulability of ET in the process, albeit with an improved probability towards ET schedulability via different heuristics [55]. In contrast, we only accept solutions in which both TT and ET tasks are schedulable, starting from the schedulability of ET tasks to impose constraints on the TT schedule generation.

Hierarchical scheduling approaches such as [1, 61, 62, 58] can be viewed as a more generalized form of a polling approach (c.f. Section 4), where on one level there is a fixed-priority scheduler for ET tasks, and on the underlying layer, a periodic resource abstraction (or periodic server) is used to decouple TT schedule generation from ET task schedulability analysis. Using the worst-case service pattern for the periodic resource abstraction, as is done in [1], has, on the one hand, the downside of the abstraction overhead (c.f. [62]) and, on the other hand, the server design problem [42] makes the problem difficult to solve, even for bandwidth-optimal approaches such as [16]. Moreover, for a mixed ET and TT system, there is the additional complexity of deciding how many polling tasks (i.e., resource abstractions/servers) to use and how to assign ET task subsets to the resources. The method proposed in [1], which we use for comparison, is designed for constrained deadline ET tasks, whereas our affine envelope approximation considers arbitrary ET deadlines. Furthermore, we use a completely different approach based on affine envelope approximation that eliminates the need for solving the server design problem and achieves a better performance in terms of schedulability and runtime compared to e.g. [1].

A third category of related work concerns the integration of TT and ET tasks at runtime, where the main goal is to minimize the response times of ET tasks. In [25, 33, 34], a slot-shifting method is presented, which allows static TT schedule slots to be moved in

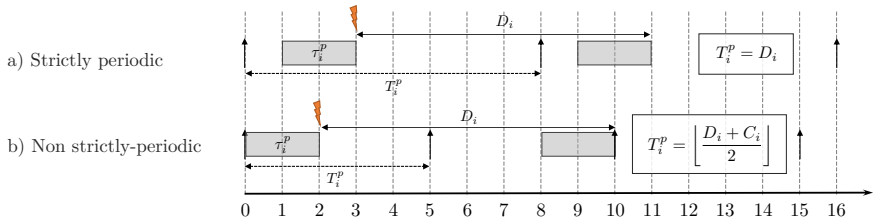


Figure 2 Worst-case event arrival wrt. slot placement.

order to execute sporadic and aperiodic tasks arriving dynamically at runtime. In [63, 64] the schedule is safely adapted at runtime to allow for improved Quality-of-Service or the execution best-effort tasks. Moreover, in [34], the authors also include an offline analysis for sporadic ET tasks, which only looks at so-called critical slots in order to guarantee schedulability. However, the main assumption is that slot shifting is possible in the TT schedule, a property not implemented in many table-driven dispatchers. The method in [56], while not directly modifying the TT schedule at runtime, has for a more flexible TT model using priority-based scheduler for ET tasks, similar to our system model, but allowing the TT schedule to be configured to an arbitrary priority in relation to the ET tasks. The work in [13] an SMT-based static schedule table synthesis is combined with an EDF-based online scheduler that handles sporadic ET tasks. While these approaches may work well for the average case, they do not mandate the schedulability of ET tasks and, additionally, impose a flexible execution model on the TT dispatcher.

4 TT and ET integration using Polling Tasks

For sporadic tasks, a straightforward approach is to simulate or analyze the worst-case behavior of sporadic ET tasks for every (or selected) possible variation of the idle slots in the static scheduling table reserved for the polling task(s), similar to [54, 55]. Hence, the schedule synthesis step has to check for any placement of the TT slots if the resulting idle slots used by the polling task to handle ET tasks are sufficient to fulfill the deadlines of ET tasks. A feedback loop could, in principle, offer some heuristic suggestions that may guide the search and, on average, speed up the algorithm. However, while heuristics like the ones provided in [54, 55] may create correct schedules for both TT and ET tasks, there is no guarantee of this since they tend to favor TT schedule correctness over ET. If ET schedulability is crucial, a brute-force approach that will check every possible TT schedule for ET schedulability is intractable and will not scale for medium and large systems. Hence, we introduce two polling-based methods inspired by prior work that can be used to guarantee sporadic ET tasks in systems with a static TT schedule table.

A simple and computationally "cheap" method, which we call *simple polling* **(SPoll)**, is to let each ET task $\tau_i \in \mathcal{T}^{ET}$ be handled by its own polling TT task τ_i^p. If we know that the generated schedule table will be strictly periodic with respect to the placement of the polling task and the deadline is equal to the period, we can set the period of the polling task τ_i^p to $T_i^p = D_i$ and the computation time to $C_i^p = C_i$ (c.f. Figure 2 (a)). In the case of non-strictly periodic slot placement (when using, e.g., EDF simulation to generate the schedule table [14]) and arbitrary ET deadlines, we have to use over-sampling to place the polling task in the static schedule table (c.f. Figure 2(b)). The over-sampling period T_p is easily derived for e.g., out of the availability function in [1] as $T_i^p = \lfloor \frac{D_i + C_i}{2} \rfloor$. For sporadic ET tasks with constrained deadlines, the polling task has a computation time $C_i^p = C_i$.

For sporadic ET tasks with arbitrary deadlines, we need to consider how many previous job releases there can be within any polling period. Hence we have $C_i^p = \lceil \frac{T_i^p}{T_i} \rceil \cdot C_i$. This approach can be very pessimistic for tasks with a short deadline and long MIT/period or a long deadline and short period/MIT, leading to a reduced schedulability. Consider a similar example to the one from [72], where a sporadic event with a computation time of $C_i = 2$ ms and a deadline of $D_i = 20$ ms needs to be handled. The event can occur at most once every $T_i = 100$ ms, thereby having a 2% CPU utilization. However, because the exact arrival is not known, we would have to reserve a slot every 9 ms (assuming a 2 ms slot size) if we want to finish the execution within 20 ms of any possible event arrival. This would consume $22,2\%$ of the CPU bandwidth. If the deadline is much larger than the period, e.g., $D_i = 100$ ms, $T_i = 10$ ms, and $C_i = 2$ ms, the period and the computation time of the polling task are 49 and 10, respectively, which results in a utilization of $20,4\%$. We see that, except in very simple systems, this approach results in a large over-utilization and will most likely not result in any feasible TT schedule creation. Both strictly periodic and non-strictly periodic approaches have the downside of reduced schedulability, either from oversampling or from the strictly periodic nature of ET slot placement.

A more precise approach, which we call *advanced polling* **(AdvPoll)**, is a simplified version of the hierarchical scheduling paradigm [61, 62, 42] with 2 levels, a 2nd-level fixed-priority (FP) dispatcher for ET tasks, and a time-triggered dispatcher on the lowest level, similar to [1]. Our reference method for the advanced polling is [1] where the schedulability of a set of constrained deadline sporadic tasks is verified under a server with a given capacity and period. Similar to [1], we define a periodic resource abstraction (basically a budget and period) for each polling task such that the sporadic ET tasks are still schedulable if the polling task gets the desired budget in the given period. The offline schedule synthesis step for TT tasks can then readily include the polling task(s) when generating the schedule table as another periodic (set of) TT task(s), e.g., using exact methods or heuristics (c.f. [60]). Naturally, there can be more than one polling task, each of them handling a disjoint subset of the ET tasks. To ease the notation, we assume for now that there is only one polling task τ_p handling the entire set $\mathcal{T}^{ET}$ of ET tasks for which C_p and T_p have to be determined. While in [1] the polling task (periodic resource) is defined by a budget C_p and a period T_p, a more general model called Explicit Deadline Periodic (EDP) [16, 2, 42] can be used in which the server also has a deadline $D_p \leqslant T_p$. While this extension may increase the search space for possible TT schedules (and therefore schedulability), it will also result in a more complex server design problem (see below). We hence use the more simple model from [1] to define the lower supply bound function $slbf(t)$ of a polling task τ^p in any time window of length $t \geqslant 0$. The exact expression of $slbf(t)$ can be found in [1], based on the characteristic function from [42]. To reduce complexity, the $slbf(t)$ is usually bound linearly from below by the so-called linear supply lower bound function $lslbf(t)$ (c.f. [42]) defined in [1] using $a = \frac{C_p}{T_p}$ and $\Delta = 2 \cdot (T_p - C_p)$, as

$$lslbf(t) = max\{0, (t - \Delta) \cdot a\}. \tag{4}$$

Following the method in [1], we compute for each ET task $\tau_i \in \mathcal{T}^{ET}$ and for each instant t the maximum load of task τ_i and all higher and equal priority tasks (maximum load of level-i) $H_i(t)$. We can use the classical definition of the maxium load of level-i from [39] for constrained deadline tasks, namely

$$H_i(t) = \sum_{j=1}^{i} \left\lceil \frac{t}{T_j} \right\rceil \cdot C_j. \tag{5}$$

The schedulability condition for an ET task $\tau_i \in \mathcal{T}^{ET}$ is defined in Lemma 1 of [1]. The worst-case response time R_i for task $\tau_i \in \mathcal{T}^{ET}$ can be calculated by determining the earliest time instant in which the maximum load of level-i $H_i(t)$ intersects the linear supply bound function $lslbf(t)$ of the polling task from Eq.(4), as follows [1]:

$$R_i = \text{earliest } t : t = \Delta + H_i(t)/\alpha. \tag{6}$$

Using this method, the parameters of the polling task need to be found (which is, in essence, the non-trivial server design problem [61, 1]) and then consider the polling task as a regular TT task alongside the other TT tasks in the system when creating the schedule table. The schedule generation step can be done relatively efficiently by simulating EDF/LLF scheduling until the hyperperiod of the TT tasks (e.g. [46, 14]), especially for harmonic TT task periods where the hyperperiods are (relatively) small [51] or when period re-dimensioning is possible to reduce the hyperperiod [11, 50]. However, the main drawback of this approach is that we have first to decide how many polling tasks to use, how to split ET tasks between them, and then for each polling task, find the computation time, the period, and the deadline. While there are specific optimizations that can be employed (e.g., using external points [1] or the methodology from [42]), the approach can be computationally intensive for large systems as the assignment of ET tasks to polling tasks is in itself a combinatorial problem.

In classical hierarchical scheduling, the aim is to find the resource abstraction with the least impact on other components, i.e., the best (C_p, T_p) where the utilization is just large enough to respect the ET deadlines. The search for the best (C_p, T_p) may be complex since we have to iterate not only through T_p, but for every T_p, we need to find C_p (and potentially D_p). We can use a simplification here in order to get rid of the binary search for C_p for every T_p since we can use the maximum C_p that does not lead to an overutilization, i.e., $C_p = \lfloor (1 - U^{TT}) \cdot T_p \rfloor$. However, we note that this optimization only applies if TT tasks have implicit deadlines, ET tasks have constrained deadlines, and if there is only one polling task with $D_p = T_p$, whereas our method also works for constrained-deadline TT tasks and arbitrary-deadline ET tasks. Hence, we use this optimization and approximation in order for the AdvPoll method to be able to compete with our approach in terms of runtime, as we will show in the experiments in Section 6.

5 TT and ET integration using affine envelope approximations

The main idea of our method is to derive a constraint on the TT schedule that will guarantee ET task schedulability and then use this constraint to build a correct TT schedule. First, the constraint is expressed as a maximal affine envelope for the TT tasks, computed such that as long as a TT schedule respects this envelope (expressed as token-bucket arrival curve), all ET tasks respect their deadlines (Section 5.1). The second step consists in building a TT schedule generation algorithm that enforces the envelope while maintaining TT task schedulability (Sections 5.2 and 5.3).

Since we know the TT task set, the utilization rate of TT tasks U^{TT} is known, but the burst b^{TT} of the linear arrival curve $\alpha^{TT}(t)$ is unknown and depends on the future schedule. Furthermore, as TT has a higher priority than ET, this burst b^{TT} impacts the ET response times. Hence, the goal of our method is first to identify the maximum burst b^{TT}_{max} such as the ET tasks fulfill their deadlines, and then to compute a TT schedule such as an arrival curve of the scheduled TT tasks is $\alpha^{TT}(t) = U^{TT} \cdot t + b^{TT}_{max}$. To do so, we first evaluate the impact of the TT tasks on the ET tasks and compute b^{TT}_{max} in Section 5.1. Then, in Sections 5.2 and 5.3 we present a scheduler capable of enforcing $\alpha^{TT}(t) = U^{TT} \cdot t + b^{TT}_{max}$.

5.1 Computing a maximal affine envelope for TT tasks

To compute the maximum TT burst such as the ET task deadlines are fulfilled, we first calculate the worst-case response time (i.e., delay) depending on the TT burst and TT utilization rate in Theorem 7, then we deduce the maximum admissible TT burst in Theorem 8. First, we can bound the TT burst as defined in Theorem 6.

▶ **Theorem 6** (Worst-case burst for TT tasks). *The function $\alpha_{U^{TT},C^{TT}}$ is an arrival curve for the set of TT tasks $\mathcal{T}^{TT}$.*

Proof. The functions α_{U_i,C_i} are arrival curves for the TT tasks τ_i. So an arrival curve for the set of TT tasks τ is $\alpha_\tau = \sum_{\tau_i} \alpha_{U_i,C_i} = \alpha_{U^{TT},C^{TT}}$ according to Proposition 1. ◀

This is a (pessimistic) burst that will be refined further. A similar result has been presented in [57, Thm. 2], but a direct proof is given here for completeness.

▶ **Theorem 7** (Response time of ET tasks). *Let $\alpha_{U^{TT},b^{TT}}$ be the arrival curve of the aggregated scheduled TT tasks, and α_{U_i,C_i} the linear arrival curve of each ET task $\tau_i \in \mathcal{T}^{ET}$. The maximum response time of a ET task τ_i of priority $p(i)$ is:*

$$hDev\left(\alpha_p^{ET}, \beta_p^{SP}\right) = \frac{b^{TT} + C_{\geqslant p}^{ET}}{\lambda - U_{>p}} \tag{7}$$

The hyperperiod does not appear in the expression since we use an overapproximation through affine functions in which we only require the individual task periods to compute this worst-case bound (e.g., [8] also have a bound independent of the hyperperiod).

Proof. Let $(\alpha_i, \ldots, \alpha_{n^{ET}})$ the linear arrival curves of the ET tasks, with $n^{ET} = \left|\mathcal{T}^{ET}\right|$. Let the arrival curves of the aggregated tasks of priority p and of priorities >p be respectively:

$$\alpha_p^{ET}(t) \stackrel{def}{=} \sum_{\tau_i \in \mathcal{T}^{ET}, p(i)=p} \alpha_i(t) \qquad \alpha_{>p}^{ET}(t) \stackrel{def}{=} \sum_{\tau_i \in \mathcal{T}^{ET}, p(i)>p} \alpha_i(t), \tag{8}$$

The online scheduler uses preemptive fixed-priority scheduling, so $\beta_p^{SP}(t)$ is a service curve for the task of priority p (cf. Theorem 4) and $hDev\left(\alpha_p, \beta_p^{SP}\right)$ is an upper bound on the delay of each task in $\mathcal{T}_p^{ET}$ (cf. Theorem 3). We also consider the linear arrival curves for ET tasks,

$$\alpha_p^{ET} = \sum_{\tau_i \in \mathcal{T}^{ET}, p(i)=p} \alpha_{U_i,C_i} \qquad \alpha_{>p}^{ET} = \sum_{\tau_i \in \mathcal{T}^{ET}, p(i)>p} \alpha_{U_i,C_i} \tag{9}$$

From the definitions in Section 2.1 and Proposition 1, we have:

$$\alpha_p^{ET} = \alpha_{U_{=p}^{ET}, C_{=p}^{ET}} \qquad \alpha_{>p}^{ET} = \alpha_{U_{>p}^{ET}, C_{>p}^{ET}} \tag{10}$$

As TT priority is higher than ET priorities, we have $U^{TT} + U_{>p}^{ET} = U_{>p}$, which gives:

$$\alpha_{>p}^{ET} + \alpha_{U^{TT},b^{TT}} = \alpha_{U_{>p}, b^{TT} + C_{>p}^{ET}} \tag{11}$$

Since all TT tasks have a higher priority than ET priority p, the sum of arrival curves with priority higher than p is $\alpha_{>p}^{ET} + \alpha_{U^{TT},b^{TT}}$. Hence, the residual service can be expressed, using Theorem 4 and Proposition 1:

$$\beta_p^{SP}(t) = \left[\lambda \cdot t - \alpha_{>p}^{ET}(t) - \alpha_{U^{TT},b^{TT}}(t)\right]_{\uparrow}^{+} = \beta_{(\lambda - U_{>p}), \frac{b^{TT} + C_{>p}^{ET}}{\lambda - U_{>p}}}(t) \tag{12}$$

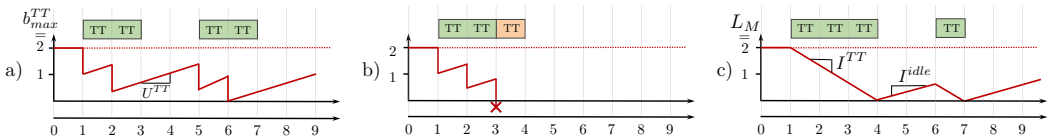


Figure 3 Checking a TT schedule under the TB and BLC constraints

Finally, with Proposition 1, we have:

$$hDev(\alpha_p^{ET}, \beta_{(\lambda - U_{>p}), \frac{b^{TT}+C^{ET}_{>p}}{\lambda - U_{>p}}}) = \frac{b^{TT} + C^{ET}_{>p}}{\lambda - U_{>p}} + \frac{C^{ET}_{=p}}{\lambda - U_{>p}} = \frac{b^{TT} + C^{ET}_{\geqslant p}}{\lambda - U_{>p}}. \tag{13}$$

◀

We now define the maximum admissible TT burst such as ET tasks fulfill their deadlines in Theorem 8.

▶ **Theorem 8** (Maximal admissible TT burst). *Let $\alpha_{U^{TT},b^{TT}}$ be the arrival curve of the aggregated scheduled TT tasks, and α_{U_i,C_i} the linear arrival curve of each ET task $\tau_i \in \mathcal{T}^{ET}$ with a priority $p(i)$. The maximum value of b^{TT} fulfilling the deadlines of all ET tasks is:*

$$b^{TT}_{max} = \min\left(\min_{p(i)}\left((\lambda - U_{>p(i)}) \cdot (\min_{\forall \tau_j, p(i)=p(j)} D_j) - C^{ET}_{\geqslant p(i)}\right), C^{TT}\right) \tag{14}$$

Proof. From Theorem 7, we know that if $\forall \tau_j$,

$$\frac{b^{TT} + C^{ET}_{\geqslant p(i)}}{\lambda - U_{>p(i)}} \leqslant \min_{p(i)=p(j)} D_j,$$

all ET tasks of priority $p(i)$ respect their deadlines. So

$$b^{TT} \leqslant \min_{p(i)}\left((\lambda - U_{>p(i)}) \cdot (\min_{p(i)=p(j)} D_j) - C^{ET}_{\geqslant p(i)}\right)$$

and we know from Theorem 6 that $b^{TT} \leqslant C^{TT}$. ◀

For methods using the lower supply bound function, i.e. minimum service curve, we note that $lslbf(t)$ (c.f. Eq. (4)) uses the same linear approximation as $\beta_p^{SP}(t)$ (c.f. Eq. (12)). However, they are built under different hypotheses. The $lslbf(t)$ is computed considering only the slot duration and period that will be assigned to the polling tasks τ_p, whereas $\beta_p^{SP}(t)$ considers the impact of higher priority tasks on the current set of ET tasks of priority p. In $lslbf(t)$ the unknown values are T_p and C_p, if we do not take the simplifications that lead to more pessimism explained in Section 4. In $\beta_{SP}(t)$ from our approach the unknown value is b^{TT}. However, after the unknown variables are computed, in both cases, the functions will lead to worst-case delays lower than the deadlines. As for the maximum requested computation, we described the ready tasks using a linear approximation $\alpha_{r,b}(t)$, instead of the more precise staircase function $H_i(t)$ defined in Eq. (5).

5.2 Burst Limiting Constraint (BLC)

Traditionally, a token bucket **(TB)** (or a leaky bucket) shaper would be used to check the computed TT envelope. Under a TB, the budget for a slot (i.e., 1) is paid at the slot allocation time t, while the budget continuously increases with a rate I^{idle} as illustrated in

Figure 3 (a). This means that the budget is a non-continuous function. However, we are trying to check a continuous function $\alpha^{TT}(t) = U^{TT} \cdot t + b^{TT}_{max}$. Hence, between the moment the budget is paid and the end of the slot, the arrival curve function we are trying to conform to has increased by $\alpha^{TT}(1) = U^{TT} + b^{TT}_{max}$. In Figure 3 (b), in interval $[1, 4]$, with a maximum burst $b^{TT}_{max} = 2$ and a replenishment rate of $U^{TT} = 1/3$, the maximum allowed cumulative processing done for TT is 3. Hence, the third TT task at $t = 3$ does conform to $\alpha^{TT}(t)$, but the TB is too pessimistic due to paying the full budget at the start of the slot. Due to this, the TB is only optimal for infinitesimally small resource demand granularity [37].

To improve this issue, we introduce a so-called *Burst Limiting Constraint* **(BLC)**, inspired by the Burst Limiting Shaper (BLS) [28, 70] and the Credit Based Shaper (CBS)[31]. Instead of paying the budget at the start of the slot, we check that the budget at the end of the slot conforms to the maximum arrival curve, and we pay the budget continuously during the slot at a rate $I^{TT} = 1 - U^{TT}$. Hence, at the end of a TT slot, the budget variation is the same as with a TB, but without the discontinuity of the budget. We detail the BLC in Definition 9, and we will show in Theorem 10 that the proposed BLC offers a maximum service curve that can be easily parameterized to fit $\alpha^{TT}(t)$.

▶ **Definition 9** (Burst Limiting Constraint). *Given a TT schedule σ, we define a Burst Limiting Constraint (BLC) such that a slot reserved for TT in σ is invalid if the budget is strictly smaller than 0 at the end of the slot, with the budget defined as follows:*

- *the budget $bdg_\sigma(t)$ is a continuous piecewise linear function of the time $t \in \mathbb{R}^+ \mapsto \mathbb{R}$,*
- *when a slot is reserved for TT in σ, the budget decreases at a rate $bdg'_\sigma(t) = -I^{TT} < 0$,*
- *when a slot is idle in σ (i.e., not assigned to TT), the budget increases at a rate $bdg'_\sigma(t) = I^{idle} > 0$ while the budget is strictly smaller than a maximum value L_M, or else it remains constant at L_M, i.e. $bdg'_\sigma(t) = 0$,*
- *the sum of I^{TT} and I^{idle} is the processing capacity: λ,*
- *at time 0, the budget is L_M, i.e. $bdg_\sigma(0) = L_M$.*

The unit of the budget is the computation unit, the unit of I^{TT} and I^{idle} is computation units per time unit. The BLC budget variations are illustrated in Figure 3 (c). We can see that in the interval $[1, 4]$, we are able to assign 3 slots, which is the maximum amount allowed by $\alpha^{TT}(t) = 1/3 \cdot t + 2$ for an interval of duration 3. In this respect, the BLC does better than the TB in Figure 3 (b). However, in the interval $[0, 6]$, with a first slot idle, there can be only 3 TT slots due to the saturation of the budget between in $[0, 1]$. So, while the BLC itself is not optimal either, its performance is better than the TB, and this difference can significantly impact schedulability. As visible in Figure 4 (b) vs. 4 (c), for a maximum burst $b = 2$ under TB, the schedule is invalid, but when transforming the TB to a BLC with the same burst (L_M), the schedule becomes valid.

While the BLC resembles the CBS and BLS by its use of a budget/credit, Definition 9 shows that it is quite different from either of them. With the BLC and contrary to the credit of the CBS [32], the budget is continuous, and we set the budget upper and lower bounds. Moreover, while the BLS has these 3 properties, with the BLC, there is no priority inversion at a defined level L_R, and there can be no saturation of the budget at 0 [22].

We now present the maximum service curve offered to TT tasks by the BLC.

▶ **Theorem 10** (BLC maximum service curve). *The maximum service curve of a set of scheduled TT tasks validated by the Burst Limiting Constraint (BLC) defined in Def. 9 is $\gamma^{TT}_{blc}(t) = I^{idle} \cdot t + L_M$.*

Proof. The proof is based on the proofs detailed in [22, 21] for the Burst Limiting Shaper (BLS) [28, 70] in TSN networks. We denote $\mathcal{C}^{TT}(t)$ the computation capacity function offered

to TT tasks, and $\Delta\mathcal{C}^{TT}(t,\delta) = \mathcal{C}^{TT}(t+\delta) - \mathcal{C}^{TT}(t)$ its variation during an interval $\delta \geqslant 0$, and λ the total processing capacity (in computation units per time unit). Hence, $\frac{\Delta\mathcal{C}^{TT}(t,\delta)}{\lambda}$ represent the executing time of the tasks TT during any interval δ. According to Definition 2, we search $\gamma^{TT}_{blc}(\delta)$ such as $\Delta\mathcal{C}^{TT}(t,\delta) \leqslant \gamma^{TT}_{blc}(\delta), \forall t \geqslant 0$.

We consider a known TT schedule σ. If σ fulfills the BLC, then we know that $bdg_\sigma(t) \geqslant 0, \forall t \geqslant 0$ and that the budget cannot saturate at 0: if the budget is 0, the next slot will be idle to fulfill the BLC and so the budget will increase. Therefore, there are three possible variations of the budget: 1) the budget increases when a slot is idle, and the budget is strictly smaller than L_M; 2) the budget decreases when a slot is assigned to TT in σ; 3) the budget saturates at L_M when a slot is idle, and the budget is already at L_M. Hence, we denote $\Delta\mathcal{C}_{L_M,sat}(t,\delta)$ the number of computation units where the budget is saturated at L_M.

We present here a lemma linked to the budget saturation and necessary for the maximum service curve proof. In Lemma 11, we show how to bound the sum of the budget consumed and the budget gained, depending on the budget saturation.

▶ **Lemma 11** (Continuous budget bounds). *$\forall$ set of assigned TT tasks fulfilling a BLC, $\forall t \geqslant 0, \delta \geqslant 0$, the variation of the computation capacity $\Delta C^{TT}(t,\delta)$ is bounded by:*

$$-L_M \leqslant -\Delta C^{TT}(t,\delta) + (\delta - \frac{\Delta C_{L_M,sat}(t,\delta)}{\lambda}) \cdot I^{idle} \leqslant L_M$$

Proof. In an interval $t, t+\delta$, for any set of assigned TT tasks fulfilling the BLC, the accurate consumed budget is the duration corresponding to the slots $\frac{\Delta\mathcal{C}^{TT}(t,\delta)}{\lambda}$ multiplied by the signed TT slope:

$$budget_{consumed} = \frac{\Delta\mathcal{C}^{TT}(t,\delta)}{\lambda} \cdot (-I^{TT}).$$

Conversely, the gained budget is the remaining time $\delta - \frac{\Delta\mathcal{C}^{TT}(t,\delta)}{\lambda}$ minus the saturation time $\frac{\Delta\mathcal{C}_{L_M,sat}(t,\delta)}{\lambda}$, multiplied by the idle slope:

$$budget_{gained} = \left(\delta - \frac{\Delta\mathcal{C}^{TT}(t,\delta) + \Delta\mathcal{C}_{L_M,sat}(t,\delta)}{\lambda}\right) \cdot I^{idle}.$$

Thus $\forall \delta \in \mathbb{R}^+$, using the fact that $I^{TT} + I^{idle} = \lambda$, the sum of the gained and consumed budget, expressed as $budget_{consumed} + budget_{gained}$, is:

$$-\Delta\mathcal{C}^{TT}(t,\delta) + \left(\delta - \frac{\Delta\mathcal{C}_{L_M,sat}(t,\delta)}{\lambda}\right) \cdot I^{idle}.$$

Since the budget is a continuous function with lower and upper bounds 0 and L_M, respectively, the sum of the consumed and gained budget is always bounded by $-L_M$ and $+L_M$:

$$-L_M \leqslant -\Delta C^{TT}(t,\delta) + \left(\delta - \frac{\Delta C_{L_M,sat}(t,\delta)}{\lambda}\right) \cdot I^{idle} \leqslant L_M$$

◀

Returning to the proof of Theorem 10, we know from Lemma 11 that $-L_M \leqslant -\Delta\mathcal{C}^{TT}(t,\delta) + (\delta - \frac{\Delta\mathcal{C}_{L_M,sat}(t,\delta)}{\lambda}) \cdot I^{idle}$. Thus,

$$\Delta\mathcal{C}^{TT}(t,\delta) \leqslant L_M + \left(\delta - \frac{\Delta\mathcal{C}_{L_M,sat}(t,\delta)}{\lambda}\right) \cdot I^{idle}.$$

We know by definition that: $\Delta\mathcal{C}_{L_M,sat}(t,\delta) \geqslant 0$. Hence, we obtain

$$\Delta\mathcal{C}^{TT}(t,\delta) \leqslant I^{idle} \cdot \delta + L_M = \gamma^{TT}_{blc}(\delta).$$

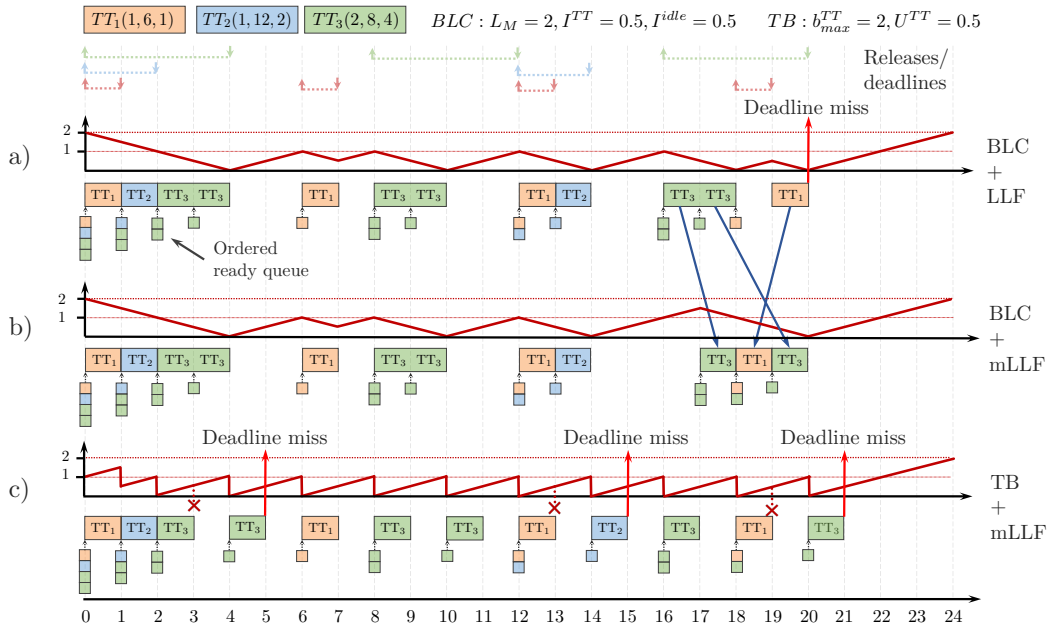


Figure 4 (a) LLF under a BLC constraint vs. (b) mLLF under a BLC constraint vs. (c) mLLF under a TB constraint.

5.3 Burst Limiting Least Laxity First (B3LF)

While checking that an existing TT schedule adheres to the BLC (or TB) is easy[1], the more interesting (and useful) question is how to create TT schedules that respect the BLC constraint. A first idea would be to emulate well-known mechanisms such as Earliest-Deadline-First (EDF) [43] or Least-Laxity-First (LLF) [41] while keeping the schedule under the BLC, as illustrated in Figure 4 (a). However, it is easy to show a counterexample (c.f. Figure 4 (a) vs. Figure 4 (b)) proving that it is not an optimal result and it may lead to deadline misses. The problem with using EDF/LLF is that it can reach the maximum allowed burst by scheduling a task immediately if, e.g., it is the only one in the ready queue, even though it has enough slack and could be executed later. Thus, at the next time instant, there is no more available budget, and we cannot schedule any 0-slack TT task that has been released, leading to a deadline miss. This problem will persist under any work-conserving algorithm since sometimes it may be necessary to insert idle times to have the full burst at a later time when it may be needed.

To solve this problem, we use the BLC to enforce the ET constraints through the maximum burst of TT, and we combine it with a modified LLF **(mLLF)** algorithm as described in Section 5.3. Together, the BLC and the mLLF algorithm result in a scheduler that enforces both TT and ET tasks deadlines. We call our method the *Burst Limiting Least Laxity First* scheduler **(B3LF)**. As we will detail in Section 5.3, the B3LF algorithm respects the proposed BLC by construction. The main idea of the following analysis is that mLLF itself does not negatively constrain the TT tasks, so by modeling the BLC, we are able to model

[1] Given a TT schedule, it is easy to check that it respects a given token-bucket constraint, as illustrated in Figure 3 (a) and (b). This can be done in linear time with regards to the schedule length, and if the token-bucket shape of a schedule is known, it can be updated in case of update of the schedule without a complete re-computation.

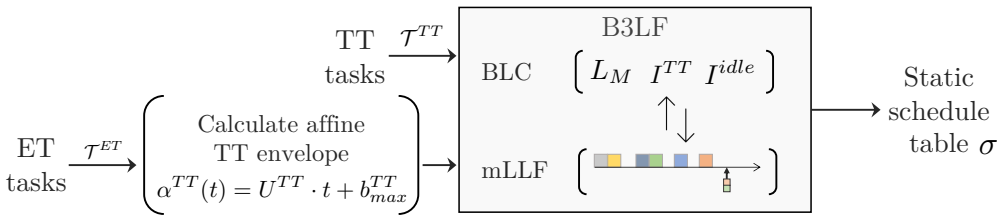


Figure 5 Burst Limiting Least Laxity First Scheduler (B3LF) and BLC parameterization

the TT constraint enforced by the whole B3LF. Hence, to model the B3LF in RTC, we separate it into its two components: the BLC and the mLLF, as illustrated in Figure 5. The B3LF is executed offline to create a static schedule table such that the TT arrival curve at runtime is $\alpha^{TT}_{sp} \leqslant \alpha^{TT}(t) = U^{TT} \cdot t + b^{TT}_{max}$, to enforce the ET deadlines (Theorem 7).

In our model, the schedule is the output of the mLLF which itself depends on the BLC. Thus α^{TT}_{sp} is limited by the maximum service curve of the B3LF $\gamma^{TT}_{b3lf}(t)$, which is the minimum of the maximum service curves of the BLC $\gamma^{TT}_{blc}(t)$ and mLLF $\gamma^{TT}_{mllf}(t)$:

$$\alpha^{TT}_{sp}(t) \leqslant \gamma^{TT}_{b3lf}(t) = \min(\gamma^{TT}_{blc}(t), \gamma^{TT}_{mllf}(t)) \tag{15}$$

However, as will be shown in Theorem 12 in Section 5.3.1, the maximum service offered by the mLLF to TT tasks is only limited by the CPU processing capacity λ. So the TT input arrival curve in SP can only be constrained under $\lambda \cdot t$ by the BLC:

$$\alpha^{TT}_{sp}(t) \leqslant \gamma^{TT}_{blc}(t) = I^{idle} \cdot t + L_M \tag{16}$$

Finally, from the ET tasks, we have computed the affine TT envelope $\alpha^{TT}(t)$ to enforce the ET deadlines. As presented previously, we set the BLC parameters $I^{idle} = U^{TT}$, $I^{TT} = \lambda - U^{TT}$ and $L_M = b^{TT}_{max}$ and so we obtain a schedule with $\alpha^{TT}_{sp}(t) \leqslant \alpha^{TT}(t) = U^{TT} \cdot t + b^{TT}_{max}$, as illustrated in Figure 5.

5.3.1 B3LF Algorithm

The BLC ensures that the ET constraints are met by enforcing the TT slot allocation according to Definition 9 with the budget replenishment and consumption rates given by I^{idle} and I^{TT}, respectively, within the budget bounds 0 and L_M. However, we also need to ensure that the TT tasks are schedulable. Employing a standard LLF (or EDF) without BLC will only result in the schedulability of the TT tasks as shown below (Theorem 12).

▶ **Theorem 12** (LLF and mLLF maximum service curve for TT tasks). *The maximum service curve of a set of TT tasks scheduled using the Least Laxity First, with (mLLF) or without (LLF) our proposed modification to add Idle tasks, for a processing capacity of λ, is:*

$$\gamma^{TT}_{llf}(t) = \gamma^{TT}_{mllf}(t) = \lambda \cdot t \tag{17}$$

Proof. If utilization $U^{TT} = \lambda$, then LLF/mLLF assigns all the slots to TT, and the full processing capacity is used by TT, so, according to, Definition 2, we have

$$\mathcal{C}(t) - \mathcal{C}(s) \leqslant \lambda \cdot (t - s) = \gamma^{TT}_{llf}(t - s) = \gamma^{TT}_{mllf}(t - s), \forall s \leqslant t.$$

◀

Hence we introduce a modified LLF scheduler (mLLF) that constructs the static schedule table under the given BLC such that the TT task deadlines are met. LLF (Least Laxity First) [41] assigns dynamic priorities according to the current task laxity, i.e., tasks with smaller laxity are assigned a higher priority. We use LLF (instead of, e.g., EDF) because it better suits our need to track the BLC budget consumption and replenishment at any instant on the discrete timeline since LLF is a job-level dynamic priority algorithm (as opposed to EDF, which keeps priorities fixed at job-level). However, we note that a modified job-level dynamic EDF algorithm may also be a viable approach. Moreover, some practical runtime issues associated with LLF (e.g., the complex implementation or runtime calculation of laxity values) are not a concern here since we only use LLF offline to generate static schedules. Additionally, the high preemption overhead associated with "thrashing" when multiple tasks have the same laxity can be mitigated either by post-processing the resulting schedule table or by directly using modifications like ELLF [29]. ELLF aims to resolve situations in which thrashing between tasks typically occurs by executing them consecutively via excluding all but one task that would thrash from consideration. ELLF can be used straightforwardly, except that the special IDLE task is always selected based on the least laxity value and is never excluded. Post-processing would analyze the resulting schedule, identify thrashing situations, and exchange execution slots of thrashing tasks to minimize preemptions.

Our mLLF scheduler (Algorithm 2) works very similar to the standard algorithm in that at each point in time t, we compute the slack (or laxity) of a task $\tau_i \in \mathcal{T}^{TT}$ as $L_i(t) = D_i(t) - C_i(t)$, where $D_i(t)$ represents the duration from time t to the next deadline of the task, and $C_i(t)$ represents the remaining computation time at time t. In addition to the TT tasks that are considered by our mLLF, we introduce a special *IDLE* task, denoted τ_{IDLE} that is responsible for introducing idle slots into the schedule σ. The computation time, period, and deadline of τ_{IDLE} are irrelevant since it will always be active and, when selected for execution, will introduce an idle slot into the schedule. Let us denote the current budget of the BLC with $bdg_\sigma(t)$. The main aspect of the idle task is its laxity computed as

$$L_{IDLE}(t) = \begin{cases} \left\lfloor \dfrac{bdg_\sigma(t)}{I^{TT}} \right\rfloor I^{TT} & \text{if } bdg_\sigma(t) < L_M - I^{idle} \\ \infty & \text{otherwise} \end{cases} \tag{18}$$

The laxity of τ_{IDLE} at some time t is the amount of time until an idle slot must be scheduled because the budget will reach 0 when scheduling only TT tasks. Hence, the closer the budget is to 0, the higher priority τ_{IDLE} becomes, but the scheduler still allows a TT task to be scheduled if necessary. In this way, we make sure that we stay within the budget constraints of the BLC but also steer the mLLF to prefer scheduling idle slots whenever the laxity of TT tasks permits it. It is interesting to note that this customization does not change the maximum service offered to TT tasks, i.e., Theorem 12 remains valid for mLLF.

The goal now is to compute a schedule σ such as $\forall t$, the budget remains between 0 and the maximum value, i.e., $L_M = b^{TT}_{\max}$, to enforce the ET deadlines. We define two helper functions: i) last_deadline($\mathcal{T}^{TT}$, T) returns the last deadline of a task within the hyperperiod T. This is the last theoretical slot that can be attributed to a TT task. After that time, the budget will only increase, which gives us a minimal value for the budget at the end of the hyperperiod; ii) $LL(t, \mathcal{T}^{TT})$: returns the TT task with least laxity and remaining computation time out of all ready TT tasks at time t.

At time 0, the current budget is set to L_M. However, at the end of the hyperperiod T, the budget $bdg_\sigma(T)$ may be lower than at the start of the hyperperiod, meaning that we may not be able to repeat the exact same schedule as in the first hyperperiod. Hence, we study

Algorithm 1 Scheduling TT tasks under the burst limiting constraint

Data: TT tasks $\mathcal{T}^{TT}$, TT utilization U^{TT}, max burst $b^{TT}_{\max}$, hyperperiod T, processing capacity λ
Result: σ

1 $I^{TT} \leftarrow \lambda - U^{TT}$; $I^{idle} \leftarrow U^{TT}$; $L_M \leftarrow b^{TT}_{\max}$;
/* Minimal value of the budget at the end of an hyperperiod */
2 min_budget=min(T - last_deadline($\mathcal{T}^{TT}$, T)$\cdot I^{idle}$, L_M);
/* We first check if a schedule can be found for the minimal budget with Algorithm 2 */
3 initial_budget $\leftarrow$ min_budget;
4 σ=schedule(initial_budget, $\mathcal{T}^{TT}$, T,L_M, I^{TT}, I^{idle});
5 **if** $\sigma \neq \emptyset$ **then**
6 returns σ; /* A schedule has been found */
7 **if** *initial_budget* $== L_M$ **then**
8 returns $\emptyset$; /* No schedule can be found */
/* We check if the schedule can be found with the maximum budget using Algorithm 2 */
9 initial_budget = L_M;
10 σ=schedule(initial_budget, $\mathcal{T}^{TT}$, T, L_M, I^{TT}, I^{idle});
11 **if** $\sigma == \emptyset$ **then**
12 returns $\emptyset$; /* No schedule can be found */
/* Finally, we compute schedules with Algorithm 2 until we find a schedule with at least as much budget at t=T as at t=0, or fail */
13 **while** $\sigma \neq \emptyset$ & *initial_budget* $> \max(min_budget, bdg_\sigma(t))$ **do**
/* set initial budget for the next iteration */
14 initial_budget = $\lfloor \frac{bdg_\sigma(t)}{I^{TT}} \rfloor \cdot I^{TT}$;
15 σ=schedule(*initial_budget*, $\mathcal{T}^{TT}$, T, L_M, I^{TT}, I^{idle});
16 **if** $\sigma == \emptyset \vee$ *initial_budget* $\leqslant$ *min_budget* **then**
17 returns $\emptyset$; /* No schedule can be found */
18 returns σ;

different hyperperiods by varying the current budget at time 0 to construct different σ. We must find a σ fulfilling the necessary condition: $bdg_\sigma(0) \leqslant bdg_\sigma(T)$ to ensure that this σ is valid $\forall t$. We also define two sufficient conditions for the schedulability and non-schedulability:

- if a schedule σ is found with the initial budget $bdg_\sigma(0)$ at the minimal final value $min_{\sigma_i} bdg_{\sigma_i}(T)$, i.e., corresponding to the number of idle times between the last deadline and T, then this schedule is valid $\forall$ t;
- if no schedule is found with a budget at time 0 at $bdg_\sigma(0) = L_M$, then no schedule exists.

Hence, in Algorithm 1, we start by checking both sufficient conditions (Lines 3 and 9) using the function schedule(*initial_budget*, $\mathcal{T}^{TT}$, T, L_M, I^{TT}, I^{idle}) defined in Algorithm 2, where a schedule is generated according the initial budget parameter. If the sufficient conditions are not fulfilled, we run Algorithm 2 with different initial budgets (Line 13), starting with an initial budget equal to the budget at T when testing the sufficient infeasibility condition. To reduce the search, we set (Line 14) the new initial budget to be a multiple of I^{TT} rather than directly $bdg_\sigma(t)$ since, due to the budget checks, as many slots can be allocated consecutively and this reduces the search space without a significant negative impact. In over 1000 test cases, we only saw one time where the solution with the proposed initial budget failed to find a schedule that was found otherwise. However, the optimization reduced the run time

Algorithm 2 Scheduling TT tasks under the BLC depending on the initial budget

Data: initial_budget, TT task set $\mathcal{T}^{TT}$, hyperperiod T, maximum budget L_M, TT slot budget I^{TT}, idle slot budget I^{idle}
Result: σ

```
current_budget ← initial_budget;
∀ τ_i ∈ T^TT: c_i ← C_i; d_i ← D_i;
t ← 0;
while t < T do
    for τ_i ∈ T^TT do
        if c_i > 0 ∧ d_i ⩾ t then
            return ∅; /* Deadline miss! */
        if t%T_i == 0 then
            /* Task release at time t */
            c_i ← C_i;
            d_i ← t + D_i;
        if c_i > 0 then
            L_i ← (d_i − t) − c_i; /* Compute laxity of task τ_i */
    if current_budget < L_M − I^idle then
        L_IDLE ← ⌊current_budget / I^TT⌋ I^TT;
    else
        L_IDLE ← T; /* We make sure τ_IDLE has the highest laxity */
    /* Check if τ_IDLE has the least laxity out of all tasks with c_i > 0 */
    if L_idle < L_i, ∀τ_i ∈ T^TT : c_i > 0 then
        /* Schedule idle slot */
        σ[t] ← idle;
        current_budget ← min(current_budget + I^idle, L_M);
    else
        /* If there is enough budget, schedule the least-laxity ready task */
        if (current_budget ⩾ I^TT) ∧ ([c_i > 0, ∀i ∈ T^TT] ≠ ∅) then
            /* Schedule least-laxity ready task */
            σ[t] ← τ_i = LL(t, T^TT);
            c_i ← c_i − 1;
            current_budget ← current_budget − I^TT;
        else
            /* Schedule idle slot */
            σ[t] ← idle;
            current_budget ← min(current_budget + I^idle, L_M);
if [c_i > 0, ∀i ∈ T^TT] then
    /* Schedule is infeasible if any TT task has c_i > 0 at this point */
    return ∅;
returns σ;
```

by up to 98.9% in some of our test cases. The algorithm ends when a solution σ is found (i.e. $bdg_\sigma(0) \leqslant bdg_\sigma(T)$), or when the final budget reaches the minimum final budget possible, $bdg_\sigma(T) \leqslant \min_{\sigma_i} bdg_{\sigma_i}(T)$, (since when no solution is found, the function of the final budget $\sigma \mapsto bdg_\sigma(T)$ is strictly decreasing from one iteration to the next).

We note that the fundamental relation between the B3LF and any method building on the hierarchical scheduling approach is that when T_p is computed in the polling approach,

the maximum load of level-i $H_i(t)$ has to consider the worst-case polling task placement, denoted with Δ (c.f. Figure 3 in [1]). On the other hand, with our method, we constraint the TT slot placement to fit a feasible ET schedule, generally leading to $b^{TT}_{max} \leqslant \Delta$. The more exact EDP model [16] may improve the schedulability of AdvPoll but will also significantly increase the complexity of solving the server design problem. However, it may be interesting the relate b^{TT}_{max} and Δ (with the extended EDP model) and maybe use b^{TT}_{max} to derive the polling period and deadline more quickly, but we leave such endeavor for future work.

5.4 Complexity analysis

We denote $n^{TT} = \left|\mathcal{T}^{TT}\right|$, $n^{ET} = \left|\mathcal{T}^{ET}\right|$, and t the number of possible schedule slots on the timeline until the schedule repeats, i.e., the schedule cycle, which is either the hyperperiod T or a multiple thereof. The complexity of finding b^{TT}_{max} is $\mathcal{O}(n^{TT}+n^{ET})$ due to the sum over $\mathcal{T}$ to find $U_{>p}$. The complexity of Algorithm 2 is the same as a regular LLF algorithm, namely $\mathcal{O}(n^{TT} \cdot log(n^{TT}))$ (sorting by laxity) for every time slot of a potentially exponential-length schedule cycle (c.f. [4]). The complexity of the B3LF (i.e. Algorithm 1) is therefore

$$\mathcal{O}\left(\frac{C^{TT}}{\lambda - U^{TT}} \cdot t \cdot n^{TT} \cdot log(n^{TT})\right)$$

because of the complexity $\mathcal{O}(\frac{C^{TT}}{\lambda - U^{TT}})$ of the while loop search Line 13 for which the schedule cycle of B3LF is upper bounded by $\lfloor \frac{L_M}{I^{TT}} \rfloor \cdot T$. We know from Theorem 8 that $L_M \leqslant C^{TT}$ and $I^{TT} = \lambda - U^{TT}$. Consequently, $\mathcal{O}(\frac{L_M}{I^{TT}}) \sim \mathcal{O}(\frac{C^{TT}}{\lambda - U^{TT}})$. Hence, for our proposed method, illustrated in Figure 5, the complexity is

$$\mathcal{O}\left(n^{ET} + \frac{C^{TT}}{\lambda - U^{TT}} \cdot t \cdot n^{TT} \cdot \log(n^{TT})\right).$$

The hyperperiod length can grow exponentially as a function of the maximum period and the number of tasks n^{TT} [27]. However, in many practical systems, the periods of TT tasks are relatively harmonic, leading to a manageable hyperperiod length (c.f. [50, 36]). For example, in a real-world use-case from the automotive domain which motivated this work, the 151 TT tasks have periods in the set $\{5, 10, 20, 40, 80\}ms$. Even without harmonic periods, we note that all methods, including SPoll and AdvPoll, that need to produce a static TT schedule have an intrinsic exponential component in the length of the schedule cycle. However, the methods do differ in the additional complexity of guaranteeing ET tasks. SPoll guarantees ET tasks by construction in constant time, but the resulting period may lead to an even quicker hyperperiod explosion, which can be mitigated by more aggressive (and therefore wasteful) oversampling to maintain a manageable cycle size. AdvPoll has an additional complexity in checking ET task deadlines via the response-time analysis for every server configuration and, depending on the implementation, the complexity of solving the server design problem within the TT schedulability space. B3LF, on the other hand, has only a linear additional complexity (of computing the affine envelope) bounded by $\mathcal{O}(n^{TT} + n^{ET})$.

5.5 Design optimization

In the design phase of a system, it is common that not all tasks are known from the beginning, and it usually takes several iterations before the final task set is defined. Other times, some tasks might be added or changed later on in the project life-cycle. If TT tasks are known, and ET tasks are in an iterative design process, recomputing a new schedule or checking whether the old one still respects the deadlines of the new ET task set may be cumbersome.

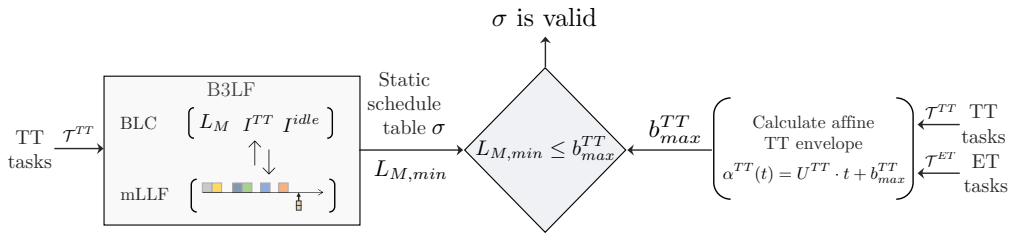


Figure 6 Design optimization of B3LF and TT affine envelop computation

With our proposed method, we can quickly check that the current ET tasks are fulfilling their deadlines by checking the b^{TT}_{max} used to compute the TT schedule. In addition to this, we propose to use a design optimization, called **BinaryB3LF**, to find the minimum L_M such that a schedule with only the TT tasks as input is feasible. With this $L_{M,\min}$, the scheduler needs to be run only once for the given set of TT tasks. Then, for each iteration of ET tasks, only the check of $b^{TT}_{max} \geqslant L_{M,\min}$ is needed to assess the fulfillment of ET deadlines without modifying the TT schedule, as illustrated in Figure 6. We know that i) b^{TT}_{max} is upper bounded by C^{TT} from Theorem 8; ii) $L_M \geqslant I_{TT}$ to be able to schedule at least one TT slot; iii) increasing L_M increases the budget available for TT in the hyperperiod T, so the schedulability of TT depending on L_M is discontinuous: not schedulable under $L_{M,\min}$, and schedulable over $L_{M,\min}$. Hence, we propose to set $I^{TT} = \lambda - U^{TT}$ and use a binary search to find the minimum value of L_M such that the TT tasks are still schedulable. The search can be limited to multiples of I^{TT} to improve runtime (see reasons explained for the *initial_budget* in Section 5.3.1).

The complexity of BinaryB3LF is

$$\mathcal{O}\left(K \cdot n^{ET} + \log\left(\frac{C^{TT} - \lambda + U^{TT}}{\lambda - U^{TT}}\right)\frac{C^{TT}}{\lambda - U^{TT}} \cdot t \cdot n^{TT}\log(n^{TT})\right),$$

with K the number of iterations done for the ET tasks, and $\mathcal{O}(\log(\frac{L_M - I^{TT}}{I^{TT}}))$ being the complexity of the binary search. Without the optimization, the complexity is $\mathcal{O}(K \cdot (n^{ET} + \frac{C^{TT}}{\lambda - U^{TT}} \cdot t \cdot n^{TT}\log(n^{TT})))$. We note that a similar design optimization can be achieved by the simplification of choosing $C_p = \lfloor(1 - U^{TT}) \cdot T_p\rfloor$ in AdvPoll described in Section 4 since computing the C_p in this way generates the most "dense" allocation for the polling task for the selected T_p, maximizing the probability of allowing ET tasks to be added or changed without recomputing the TT schedule.

6 Experiments

In this section, we compare our **B3LF** algorithm, also including the design optimization (**BinaryB3LF**), against both **SPoll** and **AdvPoll** in terms of schedulability and runtime. We implemented the SPoll method such that the polling period of each ET task does not lead to an explosion in the hyperperiod T, selecting the largest value lower than the ideal polling period that leads to a new hyperperiod that is smaller than $4 \cdot T$. The hyperperiod explosion would be significant even for small use-cases without this additional oversampling. We implemented AdvPoll as a greedy search by iterating through 200 possible equidistant polling task periods T_p up to the hyperperiod of TT tasks and for each T_p we set $C_p = \lfloor(1 - U^{TT}) \cdot T_p\rfloor$ instead of suggested binary search from [1] to reduce the complexity of the server design as explained in Section 4. We also implemented the external point optimization (Theorem 3

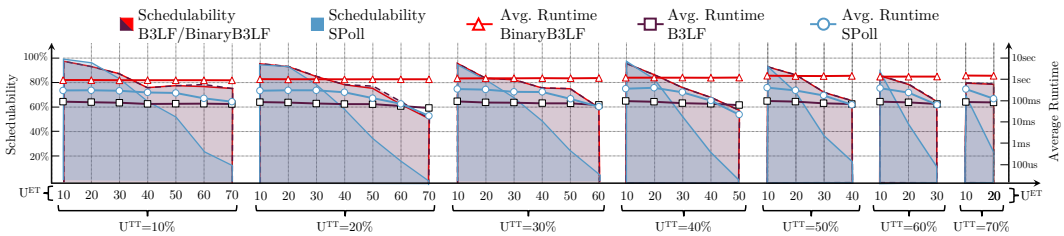


Figure 7 Schedulability and average runtime with arbitrary ET task deadlines in $[C_i, 5 \cdot T_i]$.

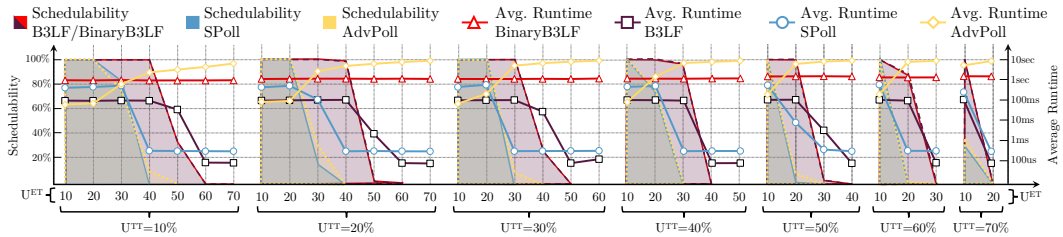


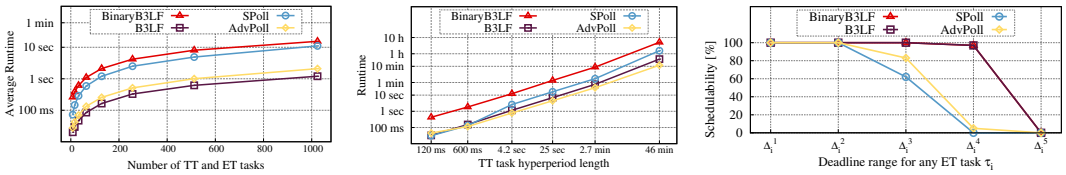
Figure 8 Schedulability and average runtime with constrained-deadline ET tasks.

of [1]) for speeding up the schedulability analysis of each (C_p, T_p) pair. Hence, we give the best possible conditions for AdvPoll to compete against our approach since its runtime is not dependent anymore on the number of possible polling task periods as it would be with a more exhaustive search. We note, however, that a more exhaustive search for T_p up to and beyond the hyperperiod and using the EDP model may improve the results but will also lead to a significant increase in the runtime. Moreover, since we assume that the TT tasks and the polling task have implicit deadlines, we also can use the efficient utilization-based test for every (C_p, T_p) candidate instead of the more complex one for constrained-deadline tasks from [4], favoring ADvPoll even more over our approach in terms of runtime. We note that the complexity of our approach does not change for constrained-deadline TT tasks. After finding the polling task(s), we use a simple LLF simulation until the hyperperiod for both SPoll and AdvPoll to generate the static TT schedule.

We extended the task set generator from [18, 19], to create task sets containing TT and ET tasks with a deadline-monotonic priority assignment between 0 and 6 for ET tasks and a priority of 7 for TT tasks. All generated task sets are schedulable if the ET tasks are considered as TT tasks and statically scheduled. All experiments were run on an Apple MacBook M1 Pro 10-core (3.12 GHz) machine with 16 GB of LPDDR5 memory.

For the first set of experiments, we compare the approaches in terms of schedulability and runtime for use-cases with a $10\mu s$ microtick, 30 TT and 20 ET tasks per task set with periods selected from the set $\{20, 30, 40\}ms$ $(T = 120ms)$, and 100 task sets per test case. For the constrained ET deadline test cases, D_i is uniformly selected in the upper half of the interval $[C_i, T_i]$, and for arbitrary ET deadlines, we use $D_i \in [T_i, 5 \cdot T_i]$. We hence construct a favorable scenario for SPoll and AdvPoll to be able to compete with our approach, as a smaller laxity will lead to worse results for the two classical methods (see Figure 9c below).

We vary $u^k \in \{0.1, 0.2, .., 0.7\}$ such that $u^{TT} + u^{ET} \leqslant 0.9$ resulting in 34 tuples (u^{TT}, u^{ET}) as seen on the x-axis of Figures 7 and 8. Our method consistently outperforms SPoll and AdvPoll in terms of schedulability (left y-axis), sometimes by a significant amount, being able to schedule over 70% of the task sets compared to under 10% for SPoll and AdvPoll. For arbitrary deadlines (Figure 7), which AdvPoll cannot handle, we achieve a high test case schedulability rate even for highly utilized systems while almost always being faster than



(a) increasing number of tasks **(b)** increasing TT hyperperiod **(c)** decreasing laxity

Figure 9 Complexity and schedulability experiments for different problem dimensions.

SPoll. For constrained-deadline systems (Figure 8), AdvPoll sometimes fares better than SPoll (but not always) but is only comparable to our approach for a low ET task utilization and low overall system utilization. For systems with higher ET task utilization, we can still schedule relatively many task sets (sometimes even 100%), while the classical methods fail to schedule any. B3LF, as well as SPoll and AdvPoll, cannot schedule any task set for systems utilizations approaching 90% or for high ET task utilization ($> 60\%$). For such systems, the main question is if a schedule that respects both TT and ET schedulability is possible at all. In almost all test cases, the schedulability of BinaryB3LF and B3LF is the same with a few exceptions where B3LF is better by $1 - 2\%$ because we consider only multiples of I^{TT} for the values of L_M in BinaryB3LF.

In terms of runtime (right logarithmic y-axis), SPoll decreases with schedulability since the algorithm ends at the first polling task, for which the oversampling leads to infeasibility. AdvPoll is only comparable in runtime with both B3LF and BinaryB3LF for poorly utilized systems where the greedy search for the server period succeeds early. As the system utilization increases and schedulability decreases, our B3LF method and AdvPoll diverge in opposite directions, meaning that while AdvPoll takes longer to give a negative result, we are very fast in identifying that we cannot schedule a certain task set. The BinaryB3LF design optimization is, as expected, slower than B3LF and SPoll (and even slower than AdvPoll in some cases) due to the binary search for the best L_M.

We did a limited series of experiments comparing the greedy AdvPoll with a brute-force AdvPoll version that tries every T_p up to the TT hyperperiod with 3 use-cases $(U^{TT}, U^{ET}) = (10, 30), (20, 30)$, and $(30, 30)$ (100 task sets each) and found that the schedulability increased from 76% to 81%, from 31% to 44%, and from 9% to 15%, respectively. The average runtime per task set increased from $0.63sec$ to $23sec$, from $2.6sec$ to $2min$, and from $5sec$ to $4.7min$, respectively. The increased schedulability was still below the schedulability of our B3LF (100% in all 3 cases), while our average runtime was around $100ms$ for all 3 cases.

In the second set of experiments, we study the runtime of our approach for increasing number of tasks (Figure 9a) and (rapidly) increasing hyperperiod (Figure 9b). Additionally, we look at the effect of decreasing laxity in constrained-deadline task sets (Figure 9c).

In Figure 9a we generate 100 constrained-deadline task sets per test case with each task set having 20% ET and 20% TT task utilization, and periods chosen from the set $\{50, 100\}ms$ to keep the hyperperiod the same across test cases. We see the effect of increasing the number of tasks (x-axis) from 8 to 1024 (equal number of TT and ET tasks) on the runtime (logarithmic y-axis) of B3LF and BinaryB3LF, confirming the theoretical complexity findings from Section 5.4 and showing the efficiency of our method in relation to SPoll and AdvPoll. AdvPoll is not impacted as much by the increasing number of tasks as perhaps intuitively expected due to the greedy search and the external points optimization from [1].

In Figure 9b we maintain the utilization setup from before and generate 1 implicit deadline task set per test case with 8 TT and 8 ET tasks, increasing the hyperperiod of TT tasks

T exponentially from $120ms$ to $2784600ms$ ($\approx 46min$) on a timeline with a $10\mu s$ microtick (logarithmic x-axis). The generation of the TT schedule dominates all other aspects when the hyperperiod explodes since all algorithms scale linearly in the number of time instants until their respective schedule cycles. Note that the schedule cycle is a function of the represented TT hyperperiod T, being either equal to it or a multiple thereof. For SPoll and AdvPoll, the schedule cycle is the lcm of T and the period(s) of the polling task(s), and for B3LF, it may be a multiple of T (c.f. Algorithm 1) upper bounded by $\lfloor \frac{L_M}{I^{TT}} \rfloor T$. The greedy AdvPoll version is somewhat faster than B3LF, but the runtime may be significantly higher in the general case since AdvPoll may need to do an exhaustive search for T_p. In B3LF, the computation of the maximum burst is independent of T. We note that even for an (unrealistically) large hyperperiod of $\approx 46min$, B3LF manages to compute a schedule table in $27min$, which is quite acceptable for an offline schedule generation tool.

In Figure 9c we maintain the setup from Figure 9a except that each task set has 8 TT and 8 ET tasks. For each ET task $\tau_i \in \mathcal{T}^{ET}$, we choose the deadline randomly in each quintile of the interval $[C_i, T_i]$ in decreasing order (x-axis), i.e., $\Delta_i^k = [T_i - k(T_i - C_i)/5, T_i - (k-1)(T_i - C_i)/5], k = 1, \ldots, 5$, leading to an increasingly smaller laxity and hence making the task sets progressively harder to schedule. While for Δ_i^1 and Δ_i^2, the schedulability of all methods is at 100%, our method fares better than SPoll and AdvPoll when the deadline of ET tasks gets more constrained for Δ_i^3 and Δ_i^4.

Finally, we note an interesting additional observation concerning the priority assignment of ET tasks. While we use a deadline-monotonic priority assignment (which is usual in practice), we noticed that our method's schedulability drops considerably with a random priority assignment. We found that this is due to Theorem 8, since for fixed ET and TT task sets, when the priority p decreases, $\lambda - U_{>p}$ and $-C^{ET}_{\geqslant p}$ decrease, so larger deadlines D_j are needed to obtain a positive b^{TT}_{max}, which is not necessarily the case with random priorities.

7 Conclusion and future work

We have addressed the integration of sporadic event-triggered (ET) tasks with arbitrary deadlines into static time-triggered (TT) schedules via a novel method based on affine envelope approximations that distills a maximal burst and rate constraint for TT tasks such that ET tasks are schedulable. Using this affine function, we introduced an LLF-based algorithm for creating static schedule tables that respect the previously computed constraint and thereby fulfill both the temporal requirements of TT and ET tasks. We have also presented an extension that enables an efficient design optimization technique for iterative design processes where ET tasks are added or changed later. We have shown through a series of synthetic test cases that our method outperforms classical polling- and hierarchical-based approaches both in terms of schedulability and runtime in most cases.

Modern applications (e.g., in the automotive domain [46, 59]) are composed of multi-core multi-SoC platforms running tasks with complex dependencies (e.g., cause-effect chains [6, 5]). In such distributed application scenarios, both the schedule generation and the task to core allocation are part of the scheduling challenge. In this paper, we focused on the schedule generation for individual cores and did not consider more complex dependencies between tasks. However, we note that our method has potential to be generalized and applied to distributed networked systems with complex dependencies between tasks. When generating the time-triggered schedule, our method effectively imposes a certain constraint on when slots for ET tasks must be inserted into the timeline (via the computed maximum burst). Hence, the task allocation and inter-dependence problems are, in essence, orthogonal to our method.

While simple dependencies between time-triggered tasks can be readily integrated into our mLLF algorithm, adding other, more complex constraints (e.g., cause-effect chains) between tasks in distributed nodes is more challenging. We envision adding the maximum burst constraint as a special constraint on TT tasks in heuristic methods like [47] and performing the maximum burst calculation for the different task-to-core allocations in swap moves of candidate solutions. Hence, we believe that our method is general enough to be applied to distributed systems with complex constraints among tasks, and we plan to investigate the integration in future work.

References

1. Luis Almeida and Paulo Pedreiras. Scheduling within temporal partitions: Response-time analysis and server design. In *Proc. EMSOFT*. ACM, 2004. doi:10.1145/1017753.1017772.
2. Amir Aminifar, Enrico Bini, Petru Eles, and Zebo Peng. Designing bandwidth-efficient stabilizing control servers. In *Proc. RTSS*. IEEE, 2013. doi:10.1109/RTSS.2013.37.
3. Sanjoy K.. Baruah, Aloysius K. Mok, and Louis E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proc. RTSS*. IEEE, 1990. doi:10.1109/REAL.1990.128746.
4. Sanjoy K. Baruah, Louis E. Rosier, and Rodney R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Syst.*, 2(4), 1990. doi:10.1007/BF01995675.
5. Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam, and Thomas Nolte. Synthesizing job-level dependencies for automotive multi-rate effect chains. In *Proc. RTCSA*, 2016. doi:10.1109/RTCSA.2016.41.
6. Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam, and Thomas Nolte. End-to-end timing analysis of cause-effect chains in automotive embedded systems. *Journal of Systems Architecture*, 80(C), 2017. doi:10.1016/j.sysarc.2017.09.004.
7. Enrico Bini and Giorgio C. Buttazzo. Schedulability analysis of periodic fixed priority systems. *IEEE Transactions on Computers*, 53(11), 2004. doi:10.1109/TC.2004.103.
8. Enrico Bini, Andrea Parri, and Giacomo Dossena. A quadratic-time response time upper bound with a tightness property. In *Proc. RTSS*, 2015. doi:10.1109/RTSS.2015.9.
9. Anne Bouillard, Marc Boyer, and Euriell Le Corronc. *Deterministic Network Calculus – From theory to practical implementation*. Wiley, 2018.
10. Marc Boyer, Pierre Roux, Hugo Daigmorte, and David Puechmaille. A residual service curve of rate-latency server used by sporadic flows computable in quadratic time for network calculus. In *Proc. ECRTS*, 2021. doi:10.4230/LIPIcs.ECRTS.2021.14.
11. Vicent Brocal, Patricia Balbastre, Rafael Ballester, and Ismael Ripoll. Task period selection to minimize hyperperiod. In *Proc. ETFA*, 2011. doi:10.1109/ETFA.2011.6059178.
12. Samarjit Chakraborty, Simon Künzli, and Lothar Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *Proc. DATE*, 2003. doi:10.1109/DATE.2003.1253607.
13. Zhuo Cheng, Jinyun Xue, Haitao Zhang, Zhen You, Qimin Hu, and Yuto Lim. Scheduling heterogeneous multiprocessor real-time systems with mixed sets of task. In *Proc. SOSE*, 2020. doi:10.1109/SOSE49046.2020.00016.
14. Silviu S. Craciunas, Ramon Serna Oliver, and Valentin Ecker. Optimal static scheduling of real-time tasks on distributed time-triggered networked systems. In *Proc. ETFA*. IEEE, 2014. doi:10.1109/ETFA.2014.7005128.
15. Calvin Deutschbein, Tom Fleming, Alan Burns, and Sanjoy K. Baruah. Multi-core cyclic executives for safety-critical systems. *Science of Computer Programming*, 172:102–116, 2019. URL: https://www.sciencedirect.com/science/article/pii/S0167642318300601, doi:https://doi.org/10.1016/j.scico.2018.11.004.

16 Arvind Easwaran, Madhukar Anand, and Insup Lee. Compositional analysis framework using edp resource models. In *Proc. RTSS*, 2007. doi:10.1109/RTSS.2007.36.

17 Petru Eles, Alex Doboli, Paul Pop, and Zebo Peng. Scheduling with bus access optimization for distributed embedded systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(5), 2000. doi:10.1109/92.894152.

18 Paul Emberson, Roger Stafford, and Robert Davis. Techniques for the synthesis of multiprocessor tasksets. In *Proc. WATERS*, 2010.

19 Paul Emberson, Roger Stafford, and Robert Davis. A taskset generator for experiments with real-time task sets. available at https://github.com/jlelli/taskgen, Accessed on 26.01.2022.

20 Rolf Ernst, Stefan Kuntz, Sophie Quinton, and Martin Simons. The Logical Execution Time Paradigm: New Perspectives for Multicore Systems (Dagstuhl Seminar 18092). *Dagstuhl Reports*, 8(2):122–149, 2018.

21 Anaïs Finzi and Ahlem Mifdaoui. Worst-case timing analysis of AFDX networks with multiple TSN/BLS shapers. *IEEE Access*, 8:106765–106784, 2020. doi:10.1109/ACCESS.2020.3000326.

22 Anaïs Finzi, Ahlem Mifdaoui, Fabrice Frances, and Emmanuel Lochin. Incorporating TSN/BLS in AFDX for mixed-criticality applications: Model and timing analysis. In *Proc. WFCS*, 2018. doi:10.1109/WFCS.2018.8402346.

23 Tom Fleming, Sanjoy K. Baruah, and Alan Burns. Improving the schedulability of mixed criticality cyclic executives via limited task splitting. In *Proc. RTNS*, 2016. doi:10.1145/2997465.2997492.

24 Tom Fleming and Alan Burns. Investigating mixed criticality cyclic executive schedule generation. In *Proc. WMC*, 2015.

25 Gerhard Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *Proc. RTSS*, 1995. doi:10.1109/REAL.1995.495205.

26 T. M. Ghazalie and Theodore P. Baker. Aperiodic servers in a deadline scheduling environment. *Real-Time Syst.*, 9(1), 1995. doi:10.1007/BF01094172.

27 Joel Goossens and Christophe Macq. Limitation of the hyper-period in real-time periodic task set generation. In *Proc. RTS*, 2001.

28 Franz-Josef Gotz. Traffic Shaper for Control Data Traffic (CDT). IEEE 802 AVB Meeting, available at https://www.ieee802.org/1/files/public/docs2012/new-goetz-CtrDataScheduler-0712-v1.pdf, Accessed on 26.01.2022.

29 Jens Hildebrandt, Frank Golatowski, and Dirk Timmermann. Scheduling coprocessor for enhanced least-laxity-first scheduling in hard real-time systems. In *Proc. ECRTS*, 1999. doi:10.1109/EMRTS.1999.777467.

30 Menglan Hu, Jun Luo, Yang Wang, Martin Lukasiewycz, and Zeng Zeng. Holistic scheduling of real-time applications in time-triggered in-vehicle networks. *IEEE Transactions on Industrial Informatics*, 10(3):1817–1828, 2014. doi:10.1109/TII.2014.2327389.

31 IEEE. 02.1Qav - Forwarding and Queuing Enhancements for Time-Sensitive Streams, 2009. URL: http://www.ieee802.org/1/pages/802.1av.html.

32 Jahanzaib Imtiaz, Jürgen Jasperneite, and Lixue Han. A performance study of ethernet audio video bridging (avb) for industrial real-time communication. In *Proc. ETFA*. IEEE, 2009. doi:10.1109/ETFA.2009.5347126.

33 Damir Isović and Gerhard Fohler. Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints. In *Proc. RTSS*, 2000. doi:10.1109/REAL.2000.896010.

34 Damir Isović and Gerhard Fohler. Handling mixed sets of tasks in combined offline and online scheduled real-time systems. *Real-Time Syst.*, 43(3), 2009. doi:10.1007/s11241-009-9088-3.

35 Herman Kopetz. Sparse time versus dense time in distributed real-time systems. In *Proc. ICDCS*, 1992. doi:10.1109/ICDCS.1992.235008.

36 Simon Kramer, Dirk Ziegenbein, and Arne Hamann. Real world automotive benchmarks for free. In *Proc. WATERS*, 2015.

37 Pratyush Kumar and Lothar Thiele. Cool shapers: Shaping real-time tasks for improved thermal guarantees. In *Proc. DAC*, 2011. doi:10.1145/2024724.2024835.

38 Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer-Verlag, Berlin, Heidelberg, 2001.

39 John Lehoczky, Lui Sha, and Ye Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *Proc. RTSS*, 1989. doi:10.1109/REAL.1989.63567.

40 John P. Lehoczky, Lui Sha, and Jay K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proc. RTSS*. IEEE, 1987.

41 Joseph Y.T. Leung. A new algorithm for scheduling periodic, real-time tasks. *Algorithmica*, 4(1):209–219, 1989. doi:10.1007/BF01553887.

42 Giuseppe Lipari and Enrico Bini. Resource partitioning among real-time applications. In *Proc. ECRTS*, 2003. doi:10.1109/EMRTS.2003.1212738.

43 C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1), 1973. doi:10.1145/321738.321743.

44 C. Douglass Locke. Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives. *Real-Time Syst.*, 4(1):37–53, 1992. doi:10.1007/BF00365463.

45 Martin Lukasiewycz, Reinhard Schneider, Dip Goswami, and Samarjit Chakraborty. Modular scheduling of distributed heterogeneous time-triggered automotive systems. In *Proc. ASP-DAC*, 2012. doi:10.1109/ASPDAC.2012.6165039.

46 Shane D. McLean, Silviu S. Craciunas, Emil A. Juul Hansen, and Paul Pop. Mapping and scheduling automotive applications on ADAS platforms using metaheuristics. In *Proc. ETFA*. IEEE, 2020. doi:10.1109/ETFA46521.2020.9212029.

47 Shane D. McLean, Emil A. Juul Hansen, Paul Pop, and Silviu S. Craciunas. Configuring ADAS platforms for automotive applications using metaheuristics. *Frontiers in Robotics and AI*, 8:353, 2022. doi:10.3389/frobt.2021.762227.

48 Anna Minaeva and Zdeněk Hanzálek. Survey on periodic scheduling for time-triggered hard real-time systems. *ACM Comput. Surv.*, 54(1), 2021. doi:10.1145/3431232.

49 Matthieu Moy and Karine Altisen. Arrival curves for real-time calculus: the causality problem and its solutions. In *Proc. TACAS*, 2010. doi:10.1007/978-3-642-12002-2_31.

50 Mitra Nasri and Gerhard Fohler. An efficient method for assigning harmonic periods to hard real-time tasks with period ranges. In *Proc. ECRTS*, 2015. doi:10.1109/ECRTS.2015.21.

51 Mitra Nasri, Gerhard Fohler, and Mehdi Kargahi. A framework to construct customized harmonic periods for real-time systems. In *Proc. ECRTS*, 2014. doi:10.1109/ECRTS.2014.31.

52 Georg Niedrist. Deterministic architecture and middleware for domain control units and simplified integration process applied to ADAS. In *Fahrerassistenzsysteme 2016*, pages 235–250. Springer Fachmedien Wiesbaden, 2018. doi:https://doi.org/10.1007/978-3-658-21444-9.

53 Paul Pop, Petru Eles, and Zebo Peng. Bus access optimization for distributed embedded systems based on schedulability analysis. In *Proc. DATE*, 2000. doi:10.1109/DATE.2000.840842.

54 Traian Pop, Petru Eles, and Zebo Peng. Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems. In *Proc. CODES*. ACM, 2002. doi:10.1145/774789.774828.

55 Traian Pop, Petru Eles, and Zebo Peng. Schedulability analysis for distributed heterogeneous time/event triggered real-time systems. In *Proc. ECRTS*, 2003. doi:10.1109/EMRTS.2003.1212751.

56 Jorge Real, Sergio Sáez, and Alfons Crespo. A hierarchical architecture for time- and event-triggered real-time systems. *J. Syst. Archit.*, 101(C), 2019. doi:10.1016/j.sysarc.2019.101652.

57 Joan Adrià Ruiz de Azua and Marc Boyer. Complete modelling of AVB in network calculus framework. In *Proc. RTNS*. ACM, 2014. doi:10.1145/2659787.2659810.

58 Saowanee Saewong, Ragunathan Raj Rajkumar, John P. Lehoczky, and Mark H. Klein. Analysis of hierarchical fixed-priority scheduling. In *Proc. ECRTS*, 2002. doi:10.1109/EMRTS.2002.1019197.

59 Florian Sagstetter, Sidharta Andalam, Peter Waszecki, Martin Lukasiewycz, Hauke Stähle, Samarjit Chakraborty, and Alois Knoll. Schedule integration framework for time-triggered automotive architectures. In *Proc. DAC*, 2014. doi:10.1145/2593069.2593211.

60 Klaus Schild and Jörg Würtz. Scheduling of time-triggered real-time systems. *Constraints*, 5(4):335–357, 2000. doi:10.1023/A:1009804226473.

61 Insik Shin and Insup Lee. Periodic resource model for compositional real-time guarantees. In *Proc. RTSS*. IEEE, 2003. doi:10.1109/REAL.2003.1253249.

62 Insik Shin and Insup Lee. Compositional real-time scheduling framework with periodic model. *ACM Trans. Embed. Comput. Syst.*, 7(3):30:1–30:39, 2008. doi:10.1145/1347375.1347383.

63 Stefanos Skalistis and Angeliki Kritikakou. Timely fine-grained interference-sensitive run-time adaptation of time-triggered schedules. In *Proc. RTSS*. IEEE, 2019. doi:10.1109/RTSS46320.2019.00030.

64 Stefanos Skalistis and Angeliki Kritikakou. Dynamic interference-sensitive run-time adaptation of time-triggered schedules. In *Proc. ECRTS*, 2020. doi:10.4230/LIPIcs.ECRTS.2020.4.

65 Brinkley Sprunt, John P. Lehoczky, and Lui Sha. Aperiodic task scheduling for hard-real-time systems. *Real-Time Syst.*, 1, 1989. doi:10.1007/BF02341920.

66 Marco Spuri and Giorgio C. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Syst.*, 10(2), 1996. doi:10.1007/BF00360340.

67 Jay K. Strosnider, John P. Lehoczky, and Lui Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers*, 44(1), 1995. doi:10.1109/12.368008.

68 Deepak Vedha Raj Sudhakar, Karsten Albers, and Frank Slomka. Generalized and scalable offset-based response time analysis of fixed priority systems. *Journal of Systems Architecture*, 112:101856, 2021. doi:https://doi.org/10.1016/j.sysarc.2020.101856.

69 Yue Tang, Yuming Jiang, Xu Jiang, and Nan Guan. Pay-burst-only-once in real-time calculus. In *Proc. RTCSA*, 2019. doi:10.1109/RTCSA.2019.8864582.

70 Sivakumar Thangamuthu, Nicola Concer, Pieter J. L. Cuijpers, and Johan J. Lukkien. Analysis of ethernet-switch traffic shapers for in-vehicle networking applications. In *Proc. DATE*, pages 55–60, 2015. doi:10.7873/DATE.2015.0045.

71 Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time calculus for scheduling hard real-time systems. In *Proc. ISCAS*, volume 4, 2000. doi:10.1109/ISCAS.2000.858698.

72 Ken Tindell and Hans Hansson. Real-time systems by fixed priority scheduling. Technical report, Uppsala University, January 1996. Course notes in Real-Time Systems. URL: https://www.it.uu.se/edu/course/homepage/realtid/ht06/Realtime_Compendium.pdf.

73 Ernesto Wandeler and Lothar Thiele. Real-time interfaces for interface-based design of real-time systems with fixed priority scheduling. In *Proc. EMSOFT*, 2005. doi:10.1145/1086228.1086246.

74 Jia Xu and David Lorge Parnas. Priority scheduling versus pre-run-time scheduling. *Real-Time Syst.*, 18(1):7–23, 2000. doi:10.1023/A:1008198310125.

75 Fengxiang Zhang and Alan Burns. Schedulability analysis for real-time systems with EDF scheduling. *IEEE Transactions on Computers*, 58(9), 2009. doi:10.1109/TC.2009.58.