Integrating Sporadic Events in Time-triggered Systems via Affine Envelope Approximations

Anaïs Finzi
TTTech Computertechnik AG
Vienna, Austria
anais.finzi@tttech.com

Silviu S. Craciunas
TTTech Computertechnik AG
Vienna, Austria
silviu.craciunas@tttech.com

Marc Boyer
ONERA / DTIS, Université de Toulouse
F-31055 Toulouse, France
marc.boyer@onera.fr

***Abstract*—We introduce a new paradigm for synthesizing time-triggered schedules that guarantees the correct temporal behavior of time-triggered (TT) tasks and the schedulability of sporadic event-triggered (ET) tasks with arbitrary deadlines at design time. The approach first expresses a constraint for the TT task schedule in the form of a maximal affine envelope that ensures 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 (BLC) and building the schedule. The BLC constraint can be added to any existing TT schedule generation method as an additional constraint on the TT slot positioning. Here, we propose an efficient TT schedule generation method that integrates the BLC constraint via simulating a modified Least-Laxity-First (LLF) scheduler. We show via synthetic and real-world test cases that our novel method achieves 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 favorable schedule for TT tasks with respect to ET schedulability, thus increasing the probability that the system remains feasible when ET tasks are later added or changed.**

I. Introduction

Time-triggered systems are routinely used in aerospace where the safety-critical nature of the applications and stringent certification requirements impose a high level of determinism [1], [2]. Recently, a survey of 120 industry practitioners working with real-time systems showed that 68% of respondents use static schedules to help improve timing predictability and 54% use time-triggered (TT) scheduling in their systems [3]. Moreover, the use of TT solutions is also gaining importance in automotive [2], [4], [5], [6] driven by the centralization of functionality onto integrated platforms (c.f. [7]) in order to support the complex real-time requirements of, e.g., advanced driver-assistance systems (ADAS) [8], [9], [10]. In particular, the complex jitter and multi-rate cause-effect requirements of ADAS applications [11], [12] cannot be easily guaranteed off-line using classical approaches and require a more predictable time-triggered architecture (TTA) [9], [13], [10]. While TTA has many benefits in terms of predictability, stability, compositionality, and determinism, the use of static schedules 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 [14], [15], [16]). Modern safety-critical systems benefit most from combining the two paradigms.

For time-triggered systems, where the schedule table is statically computed at design time, sporadic event-triggered (ET) tasks are usually 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. [17] 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 between the TT schedule generation mechanism and an ET schedulability test [18], [19] or via hierarchical scheduling [20], [21], [22], [23], [24]. For both approaches, the computational effort (besides 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 deadlines. The second step involves expressing this envelope as a burst limiting constraint (BLC) on the TT schedule and building the static schedule table. The BLC can be integrated as an additional constraint on TT slot placement into any existing method to generate TT schedules (e.g. [25], [9], [26]) for modern distributed multi-core multi-SoC platforms (c.f. [8], [9]) executing task with multi-rate chain dependencies [12], [27], [11]. We also propose an efficient method of generating TT schedules while integrating the BLC constraint via simulating a modified Least-Laxity-First (LLF) scheduler and a multi-core extension for partitioned systems. Using our novel technique, we achieve considerably better schedulability compared to the existing approaches while having lower runtimes in almost all cases. Moreover, our method enables an efficient design optimization technique for iterative design processes where ET tasks are added or changed later. Our contributions, therefore, are:

2642-7346/24/$31.00 ©2024 IEEE
DOI 10.1109/RTAS61025.2024.00010

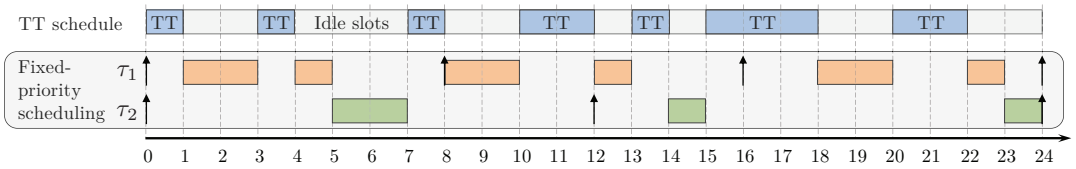


Fig. 1: Time-triggered schedule table with 2^{nd}-level fixed-priority scheduling in idle slots.

- a new and efficient approach via affine envelopes for guaranteeing the schedulability of ET tasks with arbitrary deadlines without the need for complex response-time analysis,
- a method to express the affine envelope as a burst limiting constraint (BLC) that can be integrated into any existing TT schedule synthesis algorithm,
- an LLF-based algorithm that respects the BLC constraint and generates correct TT schedules wrt. TT and ET tasks,
- 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.
- an extension for (semi-)partitioned multi-core platforms like those found in, e.g., modern automotive systems [28], [5].

We introduce some necessary preliminaries, including the system model, a short review of real-time calculus, and related work in Sec. II. We then present our novel method based on affine envelope approximations in Sec. III and evaluate it in Sec. IV. Finally, we conclude the paper in Sec. V.

II. Preliminaries

A. 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 2^{nd}-level preemptive scheduler based on fixed priorities (c.f. Fig. 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 (C_i, T_i, D_i) with C_i denoting the computation time and D_i being 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 ($D_i \leq T_i$), while ET tasks can have an arbitrary deadline, i.e., it can also be larger than the inter-arrival time. 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. Using the same pattern, we define $C^{TT} = \sum_{\tau_i \in \mathcal{T}^{TT}} C_i$, the computation time of all TT tasks. For convenience, we say that all TT tasks share the same (highest) priority. Event-triggered tasks, having a lower priority than TT tasks, are ordered and indexed according to their relative priorities, i.e., τ_i having a higher priority than τ_j ($p(i) > p(j)$) implies $i > j$. When tasks have equal priority ($p(i) = p(j)$), either FIFO or the task id can be used as a tie-breaker. Sometimes, FIFO can lead to arbitrary execution ordering when task jobs are released simultaneously; hence, a safer alternative is to use the task id as a tie-breaker. Our method works with any option.

The scheduling timeline is divided into equal segments by the microtick *mt* (also called slot length), representing the smallest scheduling granularity for tasks [29], [30]. In the following, we assume that $\forall \tau_i$, D_i, C_i, T_i are multiples of *mt*. A static schedule σ consists of a set of slots $[t \cdot mt, (t+1) \cdot mt)$ with each slot being idle or executing a TT task. Each schedule σ is repeated after the so-called hyperperiod HP (schedule cycle), which is either the least common multiple of the TT task periods or a multiple thereof (c.f. Sec. III-C). While we focus on unit-speed processors, we denote the processor capacity with the more generic λ to express that our method is also applicable to sub-unit processor speeds when, e.g., CPU frequency scaling is used (c.f. [31]).

B. Real-time (and network) calculus

Network Calculus (NC) [32] is a theory for quantifying worst-case (latency and backlog) bounds in networks. Real-time calculus (RTC) [33] is the equivalent of NC for analyzing the worst-case latency of real-time tasks (e.g. [34]). We only introduce the most important definitions and refer the reader to [33], [35] for a more in-depth description. Firstly, an arrival curve $\alpha(t)$ represents a maximum of the cumulative amount of tasks (i.e., computation time requests) that can arrive in any time interval. Secondly, a minimum (resp. maximum) service curve $\beta(t)$ (resp. $\gamma(t)$) represents a minimum (resp. maximum) of the amount of available computation time in any time interval. The response time (i.e., delay) of a task τ_i of a set of tasks $\mathcal{T}$ is detailed in Theorem 1.

Definition 1 (Arrival curve [33])**.** *An arrival curve $\alpha(t)$ of a request function $R(t)$ is a non-decreasing function which satisfies: $R(t) - R(s) \leq \alpha(t-s), \forall s \leq t$.*

Definition 2 (Service curves [33], [36], [37])**.** *A maximum service curve $\gamma(t)$ and a minimum service curve $\beta(t)$ of a capacity function $\mathcal{C}(t)$ are non-negative and non-decreasing functions satisfying: $\beta(t-s) \leq \mathcal{C}(t) - \mathcal{C}(s) \leq \gamma(t-s), \forall s \leq t$*

Theorem 1 (Maximum response time [36])**.** *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 $\beta(t)$.*

The arrival curve, minimum service curve, and maximum response time are illustrated in Fig. 2. To compute the response time of any priority, we use the service curve in Theorem 2.

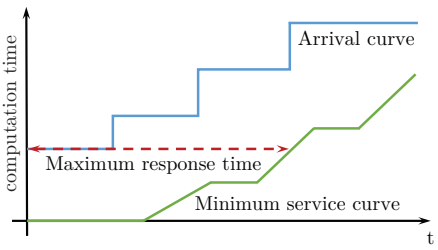


Fig. 2: Real-time calculus curves and maximum response time

Theorem 2 (Minimum remaining service curve [38])**.** *For a preemptive fixed-priority dispatcher of computation capacity λ, and a set of tasks $\tau_i \in \mathcal{T}$ with priorities $p(i)$ and arrival curves $\alpha_i(t)$, a minimum service curve remaining to tasks of priority p is the non-decreasing positive function $\beta_p^{SP}(t) = [\lambda \cdot t - \alpha_{>p}(t)]^+_\uparrow, \alpha_{>p}(t) = \sum_{\tau_i \in \mathcal{T}, p(i)>p} \alpha_i(t)$, where the notation $[]^+_\uparrow$ for a function (e.g., $g(t)$) means $[g(t)]^+_\uparrow = \left(\sup_{0 \le s \le t} g(s)\right)^+$, with $(x)^+ = \max(0, x)$.*

If a task τ_i generates jobs of cost $C_i \in \mathbb{R}^+$ at a rate given by the period (or minimal inter-arrival time) $T_i \in \mathbb{R}^+$, it admits the staircase arrival curve $\alpha : \mathbb{R}^+ \to \mathbb{R}^+$, $0 \mapsto 0$ and $t \mapsto C_i \cdot \lceil t/T_i \rceil$ if $t > 0$ [39], and the more pessimistic linear arrival curve $\alpha_{r,b} : \mathbb{R}^+ \to \mathbb{R}^+$, $0 \mapsto 0$ and $t \mapsto rt + b$ if $t > 0$, with rate $r = \frac{C_i}{T_i}$ and burst $b = r \cdot T_i$ [40]. When tasks are scheduled by a dispatcher in a certain slot, they are usually executed at a constant rate R (using one unit of computation for every time unit) after some delay (latency) L, which is due to blocking by, e.g., other higher-priority tasks. This matches a rate-latency service [40], modelled by a function $\beta_{R,L} : t \mapsto R \cdot [t - L]^+$.

Proposition 1. *Let $r, r', b, b', R, L \in \mathbb{R}^+$ be some parameters. Then, we have $\alpha_{r,b}(t) + \alpha_{r',b'}(t) = \alpha_{r+r',b+b'}(t)$.*
The proofs can be found in [39, Prop. 3.7].

C. Related work

An application model consisting of time- and event-triggered tasks is used in [18], [19], [41] for distributed embedded systems. The authors first present an analysis of periodic 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 and increase ET task schedulability. The "holistic" approach in [18], [19], [41] is 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 [19]. The method in [18], [19], [41] assumes strictly-periodic TT task slots (i.e., TT tasks execute in the same slot in each period instance), non-preemptive TT tasks, and periodic ET tasks (with bounded offsets and jitter) which greatly restricts schedulability and applicability. In contrast, we assume more generic preemptive, non-strictly-periodic TT tasks, sporadic ET tasks (which increases schedulability), and start from the schedulability of ET tasks to impose constraints on the TT schedule generation.

Meroni et al. [24] describe simple polling (*SPoll*) and advanced polling (*AdvPoll*) to integrate ET and TT tasks in a multiprocessor partitioned system. SPoll is an obvious and computationally "cheap" method that assigns to each ET task $\tau_i \in \mathcal{T}^{ET}$ its own polling TT task τ_i^p, with an oversampling period $T_i^p = \lfloor \frac{D_i + C_i}{2} \rfloor$ and computation time $C_i^p = C_i$ and $C_i^p = \lceil \frac{T_i^p}{T_i} \rceil \cdot C_i$ for constrained and arbitrary deadlines, respectively. AdvPoll is derived from hierarchical scheduling approaches such as [20], [21], [22], [23] where at one level there is a fixed-priority scheduler for ET tasks, and at the underlying layer, a periodic resource abstraction (or periodic server) is used to separate the generation of the TT schedule from the schedulability analysis of ET tasks. AdvPoll is based on the method proposed in [20], where first, the budget and period of the polling task need to be computed (a version of the server design problem [21], [20], [42]) and then the polling tasks is considered as a regular TT task alongside the other TT tasks in the system when creating the schedule table.

A third category of related work concerns adding flexibility to the runtime execution of time-triggered schedules to improve performance and allow ET task integration at runtime. In [43], [44], [45], [30], the Slot-Shifting method is presented, which allows static TT schedule slots to be moved to execute sporadic and aperiodic tasks arriving dynamically at runtime. The method assumes an existing TT schedule and provides an admission control for dynamically arriving sporadic and aperiodic ET tasks, as well as slack reclaiming to improve resource utilization. Moreover, in [45], [30], the authors also include an efficient offline analysis for known sporadic ET tasks, which only looks at so-called critical slots to guarantee ET task schedulability. In Slot-shifting, the schedulability of ET tasks is highly dependent on the heuristic to initially create and then change the TT schedule. Moreover, we note that any method requiring a recomputation of the TT schedule in case of ET infeasibility (e.g., holistic approach [18], [19] and Slot-Shifting [45], [44], [30]) will, in the general case, have a higher runtime compared to our method since we only check ET schedulability and create the TT schedule once.

III. TT and ET integration via affine envelopes

Our main idea 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, we derive a maximal affine envelope for the TT tasks expressed as token-bucket arrival curve. The envelope is computed such that as long as a TT schedule respects this envelope, all ET tasks meet their deadlines (Sec. III-A). The second step is improving this envelope and expressing it as a burst limiting constraint (BLC) that can be applied to any TT schedule generation algorithm (Sec. III-B). Finally, we propose our own TT schedule generation algorithm that enforces the BLC while maintaining TT schedulability (Sec. III-C).

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 that the ET tasks fulfill their deadlines, and then to compute a TT schedule such that 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 Sec. III-A. Then, in Sec. III-B and III-C we present a scheduler capable of enforcing $\alpha^{TT}(t) = U^{TT} \cdot t + b^{TT}_{max}$.

A. Computing a maximal affine envelope for TT tasks

To compute the maximum TT burst so that 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 4, then we deduce the maximum admissible TT burst in Theorem 5. First, we can bound the TT burst as defined in Theorem 3.

Theorem 3 (Worst-case burst for TT tasks)**.** *The function $\alpha_{U^{TT},C^{TT}}$: $\mathbb{R}^+ \to \mathbb{R}^+$, $0 \mapsto 0$ and $t \mapsto U^{TT} \cdot t + C^{TT}$ if $t > 0$, 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.

Theorem 4 (Response time of ET tasks)**.** *Let $\alpha_{U^{TT},b^{TT}}$ be a linear arrival curve of the aggregated scheduled TT tasks, and $\alpha^{ET}_{>p(i)}$ an arrival curve of the aggregated ET tasks with a priority strictly greater than $p(i)$. The maximum response time of an ET task τ_i of priority $p(i)$ is:*

$$hDev\left(\alpha^{ET}_{p(i)}, \left[(\lambda - U^{TT}) \cdot t - \alpha^{ET}_{>p(i)} - b^{TT}\right]^+_\uparrow\right)$$

Proof. This is a direct application of Theorems 1 and 2 with $\alpha_{U^{TT},b^{TT}}(t) = U^{TT} \cdot t + b^{TT}$. □

We now define the maximum admissible TT burst such that ET tasks fulfill their deadlines in Theorem 5.

Theorem 5 (Maximal admissible TT burst)**.** *Let $\alpha_{U^{TT},b^{TT}}$ be a linear arrival curve of the aggregated scheduled TT tasks, $\alpha^{ET}_{p(i)}$ an arrival curve of the aggregated ET tasks of priority $p(i)$, and $\alpha^{ET}_{>p(i)}$ an arrival curve of the aggregated ET tasks of priority strictly higher than $p(i)$.*

The maximum value of b^{TT} fulfilling the deadlines of all ET tasks, denoted b^{TT}_{max}, is, if it exists, the maximum value of b^{TT} such that, $0 < b^{TT} \leq C^{TT}$ and, $\forall$ ET priority $p(i)$:

$$\min_{\substack{\forall \tau_j \\ p(j)=p(i)}} (D_j) \geq hDev\left(\alpha^{ET}_{p(i)}, \left[(\lambda - U^{TT}) \cdot t - \alpha^{ET}_{>p(i)} - b^{TT}\right]^+_\uparrow\right)$$

Proof. We know from Theorem 3 that $b^{TT} \leq C^{TT}$. Additionally, $\forall$ ET priority $p(i)$, the maximum response time computed in Theorem 4 for priority $p(i)$, must be smaller than or equal to the deadlines of the tasks of priority $p(i)$. □

While it is possible to use the linear approximation of the ET arrival curves to directly calculate b^{TT}_{max}, the value found with such an approximation is much too pessimistic. In this paper, we use **staircase ET arrival curves** in Theorem 5. Hence, as we have a minimum and a maximum bound for b^{TT}, we use a binary search to find b^{TT}_{max}, but other search methods are also possible.

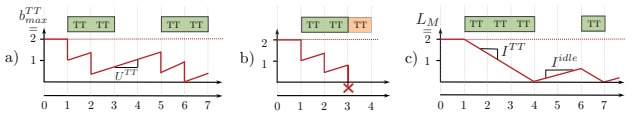


Fig. 3: Checking a TT schedule under the TB and BLC constraints

B. Burst Limiting Constraint (BLC)

Traditionally, a token bucket **(TB)** (or a leaky bucket) shaper would be used to check the computed TT envelope. As illustrated in Fig. 3 (a), the budget for a slot (i.e., 1) is paid at the slot allocation time t (an allocated TT slot is a green rectangle) under a TB, while the budget continuously increases with a rate I^{idle}. Therefore, the budget is a non-continuous function (represented by the red line) while we are trying to check a continuous function $\alpha^{TT}(t) = U^{TT} \cdot t + b^{TT}_{max}$. In Fig. 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 slot at $t = 3$ (i.e., the orange rectangle) 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 demand granularity [46].

To improve this, we introduce a so-called *Burst Limiting Constraint* **(BLC)**, inspired by the Burst Limiting Shaper (BLS) and the Credit Based Shaper (CBS) [47]. 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 $I^{TT} = \lambda - U^{TT}$ ($\lambda = 1$ for single-core unit-speed CPU). 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 3, and we show in Theorem 6 that the proposed BLC offers a maximum service curve that can be parameterized to fit $\alpha^{TT}(t)$.

Definition 3 (Burst Limiting Constraint)**.** *A slot reserved for TT in a TT schedule σ is invalid under the Burst Limiting Constraint (BLC) 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 Fig. 3 (c). We can see that in the interval $[1, 4]$, we are able to assign 3 slots, which is the max amount allowed by $\alpha^{TT}(t) = 1/3 \cdot t + 2$ for an interval of length 3. In this respect, the BLC does better than the TB in Fig. 3 (b). However, in the interval $[0, 6]$, with the first slot being idle, there can be only 3 TT slots, instead of the expected 4 (i.e., $6 \cdot 1/3 + 2$), due to the saturation of the budget in the interval $[0, 1]$. 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 Fig. 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 CBS and BLS by its use of a budget/credit, Definition 3 shows that it is quite different. Contrary to the CBS credit [48], the BLC budget is continuous, and we set the budget upper and lower bounds. Moreover, unlike BLS, with BLC, there is no priority inversion at a defined level L_R, and no saturation of the budget at 0, as is the case for the BLS [49]. We now present a maximum service curve offered to TT tasks by the BLC.

Theorem 6 (BLC maximum service curve)**.** *A maximum service curve of a set of scheduled TT tasks validated by the Burst Limiting Constraint (BLC) defined in Definition 3 is:* $\gamma^{TT}_{blc}(t) = I^{idle} \cdot t + L_M$.

Proof. The proof is based on the proofs detailed in [49], [50] for the Burst Limiting Shaper (BLS) [51], [47] 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 $\delta \geq 0$, and $\lambda = 1$ the processing capacity (in computation units per time unit). Hence, $\frac{\Delta\mathcal{C}^{TT}(t,\delta)}{\lambda}$ represents the executing time of the tasks TT during any interval δ. According to Definition 2, we search $\gamma^{TT}_{blc}(\delta)$ such that

$$\Delta\mathcal{C}^{TT}(t, \delta) \leq \gamma^{TT}_{blc}(\delta), \forall t \geq 0.$$

We consider a known TT schedule σ. If σ fulfills the BLC, we know that $bdg_\sigma(t) \geq 0, \forall t \geq 0$ and that the budget cannot saturate at 0: if the budget is 0, the next slot will be idle to fulfill the BLC; hence the budget will increase. Therefore, there are 3 possible variations of the budget: 1) the budget increases for idle slots if it is strictly smaller than L_M; 2) the budget decreases for TT slots 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 computation capacity of the time units where the budget is saturated at L_M.

We present here a lemma regarding the budget saturation that is required for the proof of the maximum service curve. Similar to [50], we show in Lemma 1 how to bound the sum of the consumed and gained budget depending on the budget saturation.

Lemma 1 (Continuous budget bounds)**.** $\forall$ *set of assigned TT tasks fulfilling a BLC,* $\forall t \geq 0, \delta \geq 0$*, the variation of the computation capacity* $\Delta C^{TT}(t, \delta)$ *is bounded by:*

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

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, similarly to [50], 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 $+L_M$:

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

□

Returning to the proof of Theorem 6, we know from Lemma 1 that

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

Thus,

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

We know by definition that: $\Delta\mathcal{C}_{L_M,sat}(t, \delta) \geq 0$, thus

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

While checking that an existing TT schedule adheres to the BLC (or TB) is easy (c.f. Fig. 3), we need to create TT schedules that respect the BLC constraint. For each time instant, the BLC essentially expresses how many consecutive TT slots (under the current BLC budget) can be in a TT schedule before an idle slot reserved for ET tasks has to be inserted into the timeline to ensure the schedulability of ET tasks. Hence, this BLC constraint can be integrated as an additional constraint into any existing TT schedule synthesis method (e.g. [25], [9], [26]) that also considers complex dependencies (e.g., multi-rate ADAS chains [12], [27], [11]) and network communication requirements (e.g. via TSN [9], [52]). We note that tracking the BLC budget consumption and replenishment is only done at design-time when creating the TT schedule so there is no runtime overhead. In the next section, we provide our own algorithm called *Burst Limiting Least Laxity First* **(B3LF)** for generating TT schedules while respecting the BLC. We believe B3LF is highly suited for the given problem, but note that our approach is not limited to it and can be applied to any TT schedule generation method.

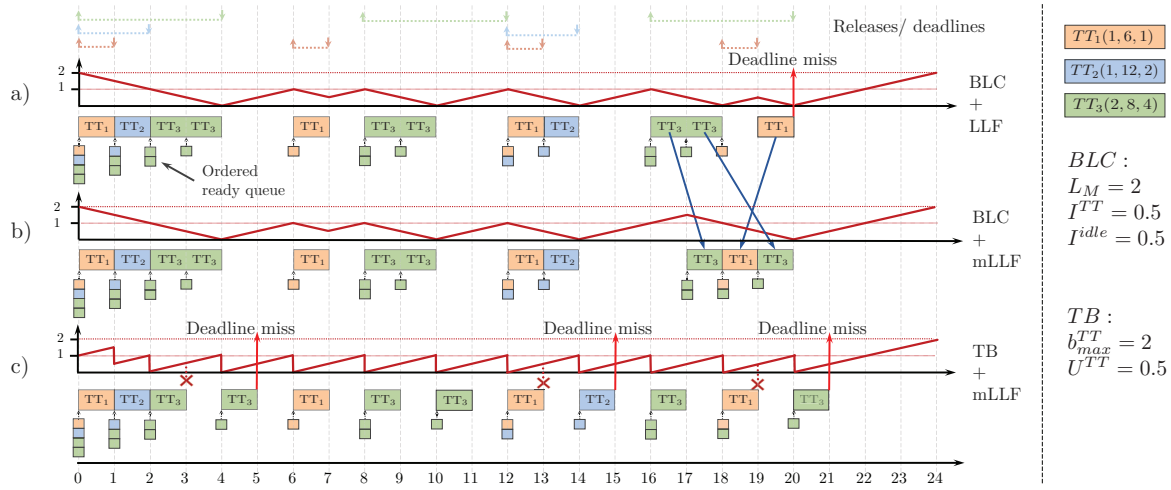


Fig. 4: (a) LLF under a BLC constraint vs. (b) mLLF under a BLC constraint vs. (c) mLLF under a TB constraint.

C. Burst Limiting Least Laxity First (B3LF)

Simulating well-known mechanisms like Earliest-Deadline-First (EDF) [53] or Least-Laxity-First (LLF) [54] has been widely used to generate static schedule tables (e.g., [55], [28]). However, we show a counterexample (c.f. Fig. 4 (a) vs. Fig. 4 (b)) where such an approach is not optimal under the BLC, leading to deadline misses. The problem with EDF/LLF under the BLC is that they can reach the maximum burst by scheduling a task immediately, e.g., if it is the only one in the ready queue, even though it has enough slack and could be executed later. Thus, at the next slot, there is no more available budget, and any 0-slack TT task that has been released cannot be scheduled, leading to a deadline miss (c.f. Fig. 4 (a)). 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 combine the BLC with a modified LLF **(mLLF)** algorithm (Sec. III-C1). Together, the BLC and the mLLF algorithm result in a scheduler that we call *Burst Limiting Least Laxity First* scheduler **(B3LF)** (Alg. 1) that enforces both TT and ET task deadlines. B3LF respects the proposed BLC by construction.

First, we show that mLLF itself does not negatively constrain the TT tasks, so by modeling the BLC, we are able to model 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 Fig. 5. The B3LF creates a TT schedule table such that the TT arrival curve at runtime is $\alpha^{TT}_{sp} \leq \alpha^{TT}(t) = U^{TT} \cdot t + b^{TT}_{max}$, to enforce ET deadlines (Theorem 5). In our model, the schedule is the output of the mLLF, which itself depends on the BLC. Thus α^{TT}_{sp} is limited by a maximum service curve of the B3LF $\gamma^{TT}_{b3lf}(t)$, which is the minimum of maximum service curves of the BLC $\gamma^{TT}_{blc}(t)$ and mLLF $\gamma^{TT}_{mllf}(t)$:

$$\alpha^{TT}_{sp}(t) \leq \gamma^{TT}_{b3lf}(t) = \min(\gamma^{TT}_{blc}(t), \gamma^{TT}_{mllf}(t)).$$

However, as will be shown in Theorem 7, our computed lower

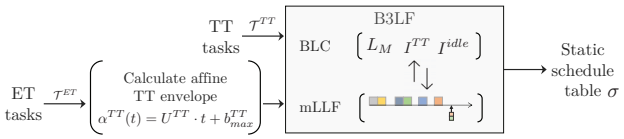


Fig. 5: B3LF and BLC parameterization

bound of the maximum service offered by the mLLF to TT tasks is only limited by the CPU capacity $\lambda = 1$. So, to shape the TT input arrival curve in strict priority (SP), we choose to use the BLC:

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

From the ET tasks, we have computed the affine TT envelope $\alpha^{TT}(t)$ to enforce the ET deadlines. Then, 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 $\alpha^{TT}_{sp}(t) \leq \alpha^{TT}(t) = U^{TT} \cdot t + b^{TT}_{max}$, as shown in Fig. 5.

The BLC ensures that ET constraints are met by enforcing the TT slot allocation according to Definition 3 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 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 7).

Theorem 7 (Lower bound of LLF and mLLF maximum service curve for TT tasks)**.** *The lower bound of the maximum service offered to 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: $\gamma^{TT}_{llf}(t) = \gamma^{TT}_{mllf}(t) = \lambda \cdot t$.*

Proof. If the utilization $U^{TT} = \lambda$, then LLF/mLLF assigns all the slots to TT, and the full processing capacity is used by TT. In the case of mLLF, the TT duration and deadlines also have to be carefully selected so there is always a TT task that has 0 laxity to prevent the idle task from being selected if its laxity reaches 0. So $\mathcal{C}(t) - \mathcal{C}(s) \leq \lambda \cdot (t - s) = \gamma^{TT}_{llf}(t - s) =$

Algorithm 1: TT schedule generation under the BLC

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

1 $I^{TT} \leftarrow \lambda - U^{TT}$; $I^{idle} \leftarrow U^{TT}$; $L_M \leftarrow b^{TT}_{\max}$;
2 $min\_budget = \min(HP - last\_deadline(\mathcal{T}^{TT}, HP) \cdot I^{idle}, L_M)$; /* Min budget at end of HP */
3 $initial\_budget \leftarrow min\_budget$; /* Can we find a schedule for the minimal budget? */
4 $\sigma = mLLF(initial\_budget, \mathcal{T}^{TT}, HP, L_M, I^{TT}, I^{idle})$;
5 **if** $\sigma \neq \emptyset$ **then**
6 return σ; /* A schedule has been found */
7 **if** $initial\_budget == L_M$ **then**
8 return $\emptyset$; /* No schedule can be found */
9 $initial\_budget = L_M$; /* Can we find a schedule for the maximal budget? */
10 $\sigma = mLLF(initial\_budget, \mathcal{T}^{TT}, HP, L_M, I^{TT}, I^{idle})$;
11 **if** $\sigma == \emptyset$ **then**
12 return $\emptyset$; /* No schedule can be found */
 /* Finally, we compute schedules with mLLF Sec. III-C1 until we find a schedule with at least as much budget at t=HP as at t=0, or fail */
13 **while** $\sigma \neq \emptyset \wedge initial\_budget > \max(min\_budget, bdg_\sigma(t))$ **do**
14 $initial\_budget = \lfloor \frac{bdg_\sigma(t)}{I^{TT}} \rfloor \cdot I^{TT}$; /* set initial budget for the next iteration */
15 $\sigma = mLLF(initial\_budget, \mathcal{T}^{TT}, HP, L_M, I^{TT}, I^{idle})$;
16 **if** $\sigma == \emptyset \vee initial\_budget \leq min\_budget$ **then**
17 return $\emptyset$; /* No schedule can be found */
18 return σ;

Algorithm 2: mLLF algorithm

Data: initial_budget, $\mathcal{T}^{TT}$, hyperperiod HP, max. budget L_M, TT slot budget I^{TT}, idle slot budget I^{idle}

1 $t \leftarrow 0$; $bdg_\sigma \leftarrow initial\_budget$; $\forall\ \tau_i \in \mathcal{T}^{TT}$: $c_i \leftarrow C_i$; $d_i \leftarrow D_i$;
2 **while** $t < HP$ **do**
3 **for** $\tau_i \in \mathcal{T}^{TT}$ **do**
4 **if** $(c_i > 0 \wedge t \geq d_i)$ **then** return $\emptyset$; /* Deadline miss */
5 **if** $(t \ \%\ T_i == 0)$ **then** $c_i \leftarrow C_i$; $d_i \leftarrow t + D_i$; /* Release at time t */
6 **if** $(c_i > 0)$ **then** $L_i \leftarrow d_i - t - c_i$; /* Laxity of τ_i */
7 **if** $bdg_\sigma < L_M - I^{idle}$ **then**
8 $L_{ID} \leftarrow \lfloor \frac{bdg_\sigma}{I^{TT}} \rfloor$;
9 **else**
10 $L_{ID} \leftarrow HP$; /* τ_{ID} gets the highest laxity */
11 **if** $L_{ID} < L_i, \forall \tau_i \in \mathcal{T}^{TT} : c_i > 0$ **then**
12 $\sigma[t] \leftarrow \tau_{ID}$; /* Schedule idle slot if τ_{ID} has least laxity of all ready tasks. */
13 $bdg_\sigma \leftarrow \min(bdg_\sigma + I^{idle}, L_M)$;
14 **else**
15 **if** $\left(bdg_\sigma \geq I^{TT}\right) \wedge \left(\left[c_i > 0, \forall \tau_i \in \mathcal{T}^{TT}\right] \neq \emptyset\right)$ **then** /* Schedule least-laxity ready task if there is enough budget */
16 $\sigma[t] \leftarrow \tau_i = LL(t, \mathcal{T}^{TT})$; $c_i \leftarrow c_i - 1$; $bdg_\sigma \leftarrow bdg_\sigma - I^{TT}$;
17 **else**
18 /* Schedule idle slot */ $\sigma[t] \leftarrow \tau_{ID}$; $bdg_\sigma \leftarrow \min(bdg_\sigma + I^{idle}, L_M)$;
19 $t \leftarrow t + 1$;
20 return σ;

$\gamma^{TT}_{mllf}(t-s), \forall s \leq t$, according to Definition 2. Hence, $\lambda \cdot t$ is a maximum service curve, and any lower service would not be a maximum service offered to TT tasks by LLF/mLLF, as shown by this counter-example. □

The goal is now to compute a schedule σ which will repeat indefinitely such that $\forall t$, the budget remains between 0 and the maximum value $L_M = b^{TT}_{\max}$, to enforce the ET deadlines. We define the helper function `last_deadline`$(\mathcal{T}^{TT}, HP)$, which returns the last deadline of a TT task within the hyperperiod HP. 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 HP. At time 0, the current budget is set to L_M. However, at the end of the first hyperperiod HP, the budget $bdg_\sigma(HP)$ may be lower than at the start of the hyperperiod, meaning that we may not be able to repeat the same TT schedule as in the first hyperperiod and still fulfill the BLC if the schedule keeps consuming more budget than is gained. Depending on σ, this may be due to the saturation of the budget at L_M. However, for the same schedule σ, as the budget at the start of a hyperperiod decreases, the saturation duration also decreases, and more budget is gained. The schedule can only be repeated indefinitely if, at some point, the sum of the budget gained in a hyperperiod is greater than the budget consumed during the same hyperperiod while still fulfilling the BLC. Consequently, we must find σ and $bdg_\sigma(0)$ fulfilling the necessary condition: $bdg_\sigma(0) \leq bdg_\sigma(HP)$ to ensure that this σ is valid $\forall t$. We also define two sufficient conditions for schedulability and non-schedulability:

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

Hence, in Alg. 1, we start by checking both sufficient conditions (lines 3 & 9) using our mLLF scheduler (c.f. Alg. 2 in Sec. III-C1) via function `mLLF`$(initial\_budget, \mathcal{T}^{TT}, HP, L_M, I^{TT}, I^{idle})$, where a schedule is generated according to the initial budget parameter. If the conditions are not fulfilled, we run the mLLF scheduler with different initial budgets (Line 13), starting with an initial budget equal to the budget at HP. 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 this reduces the search space without a significant negative impact. In some test cases (c.f. Sec. IV), the schedulability was reduced slightly, while the runtime was reduced significantly, sometimes by up to 98.9%. Alg. 1 ends when a solution σ is found (i.e., $bdg_\sigma(0) \leq bdg_\sigma(HP)$), or when the final budget reaches the minimum final budget possible, $bdg_\sigma(HP) \leq \min\_budget(HP)$, since when no solution is found, the function of the final budget $\sigma \mapsto bdg_\sigma(HP)$ is strictly decreasing from one iteration to the next.

1) mLLF algorithm: Least Laxity First) (LLF) [54] assigns dynamic priorities according to the current task laxity. 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). 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 due to "thrashing" can be mitigated either by directly using modifications like ELLF [56] or by post-processing, shifting thrashing slots to create as many contiguous same-TT task slots.

Our mLLF scheduler (Alg. 2) works very similarly 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_{ID}, that is responsible for introducing idle slots into the schedule σ. The WCET, period, and deadline of τ_{ID} are irrelevant since the special idle task will always be active and, when selected, will introduce an idle slot into the schedule. Let us denote the current budget of the BLC with $bdg_\sigma(t)$, where each time the mLLF scheduler is called from Alg. 1, the current budget is initialized to the given initial_budget value. The main aspect of the idle task is its laxity, computed as

$$L_{ID}(t) = \begin{cases} \max\left(0, \left\lfloor \frac{bdg_\sigma(t)}{I^{TT}} \right\rfloor\right) & \text{if } bdg_\sigma(t) < L_M - I^{idle} \\ \text{HP} & \text{otherwise} \end{cases}$$

The laxity of τ_{ID} at 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. If there is enough budget (i.e., $bdg_\sigma(t) \geq I^{TT}$), we schedule the TT task with the least laxity and set $bdg_\sigma = bdg_\sigma - I^{TT}$. Else, we schedule an idle slot and set $bdg_\sigma = \min(bdg_\sigma + I^{idle}, L_M)$. Hence, the closer the budget is to 0, the higher priority τ_{ID} gets, but the scheduler still allows a TT task to be scheduled if necessary. Thus, we stay within the budget constraints of the BLC but also steer the mLLF to prefer placing idle slots whenever the laxity of TT tasks permits it. Interestingly, this customization does not change the lower bound of the maximum service offered to TT tasks defined in Theorem 7, which is also valid for mLLF.

D. Design optimization

In the design stage of a system, it is common that some tasks might be added or changed later during 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. We propose to use a design optimization called **LMminB3LF**. First, using only the TT tasks as input, we find the minimum L_M, denoted $L_{M,\min}$, such that a schedule σ is found in Alg. 1 and we store ($L_{M,\min}$, σ). This schedule σ has the minimum impact on ET schedulability that is achievable with our method. Then, for each iteration of ET tasks (TT tasks are not modified, so σ is unchanged), we need to run the RTC analysis once (Theorem 4) with $b^{TT} = L_{M,\min}$ to check whether all the ET deadlines are fulfilled with σ. We know that i) b^{TT}_{max} is upper bounded by C^{TT} from Theorem 5; ii) $L_M \geq I_{TT}$ to be able to schedule at least one TT slot; iii) increasing L_M increases the budget available for TT in the hyperperiod HP, 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 $I^{TT} = \lambda - U^{TT}$, ($\lambda = 1$ for single-core unit-speed CPU) and use a binary search to find the minimum value of L_M such as Alg. 1 finds a schedule σ. The search can be limited to multiples of I^{TT} to improve runtime (see reasons in Sec. III-C).

This design optimization can also be applied to any TT schedule synthesis that includes the BLC constraint. Moreover, we note that there may exist $L_M^{'} > L_M$ with no valid TT schedule, i.e., Alg. 1 is not optimal. However, the L_M found via the design optimization is not the global minimum but the smallest out of a set of values sampled via binary search. After we found this (local-minimum) L_M, the TT schedule does not change since the design optimization is for ET tasks only, and TT tasks do not change, and no new TT tasks are added. If ET tasks are changed or added, it is irrelevant if Alg. 1 is non-optimal since we do not call Alg. 1 again; we just check ET schedulability against the original L_M.

E. Multicore and dependencies

The discussion so far has focused on single-core systems; however, our method can be easily applied to distributed multi-core systems. Creating TT schedules for modern distributed multi-SoC multi-core platforms (c.f. [8], [9], [24]) with complex dependencies between tasks involves first solving the allocation problem resulting in partitioned or semi-partitioned solutions. Then, for each core, a TT schedule that fulfills a large number of constraints needs to be found. Our method effectively imposes an additional specific constraint on when slots for ET tasks must be inserted into the timeline and, therefore, constitutes a necessary condition for the creation of schedules for TT tasks that also ensures the schedulability of ET tasks. Hence, our method is orthogonal to the multi-core task allocation and scheduling dimensions. For example, the burst limiting constraint calculated with our method can be readily added as an additional constraint on TT tasks for the different task-to-core allocations in swap moves of candidate solutions, e.g., in heuristic methods like [25], [9], [26]. Our LLF-based algorithm for generating time-triggered schedules can be easily extended with a task-to-core allocation step and hence applied to partitioned solutions in multi-core systems. Moreover, we can include TT task dependencies to, e.g., a TSN/TTEthernet network (a usual use-case in TT systems [9]) by altering the deadlines/releases of TT tasks in our mLLF algorithm when creating the schedule to fulfill dependencies.

Here, we present a simple heuristic that considers a fully partitioned solution for both TT and ET tasks. Since TT tasks are dispatched according to a static schedule, no dynamic migration decisions are being made at runtime. We also consider a fully partitioned model for ET tasks and will handle the global scheduling approach for ET tasks in future work,

noting that a (semi-)partitioned approach can achieve near-optimal schedulability [57]. A semi-partitioned solution could involve migrating TT tasks at certain predefined points (both task- and job-level) as long as the computed TT schedules include the migration overhead. The simple heuristic we used consists in sorting the ET and TT tasks in ascending order of laxity (i.e., $T_i - C_i$) and then assigning the tasks to the cores in a round-robin way. If assigning a task to a core would cause an overflow of the capacity of this core, then the core is skipped. Consequently, all cores contain a mix of low to high laxity, which increases the chance of the tasks being schedulable on each core. Please note that this simple allocation method is meant to show the applicability of our method to partitioned multi-core systems and not the effectiveness of the allocation heuristic itself.

IV. Experiments

We compare our **B3LF** algorithm, including the design optimization (**LMminB3LF**), against **SPoll** [24], **AdvPoll** [24], and **Slot-Shifting** [45], [30] in terms of schedulability and runtime using an extensive set of experiments with over 50000 data points for each method.

We implemented the SPoll method as described in [24] with the polling period of each ET task being chosen such that it does not lead to a hyperperiod explosion. We select the largest possible polling period that is lower than the ideal polling period and for which the resulting hyperperiod does not increase beyond $4HP$. We implemented AdvPoll as described in [24] using a greedy search that iterates through 200 equidistant polling task periods T^p from the interval $[1, HP]$. We also directly set $C^p = \lfloor (1 - U^{TT}) \cdot T^p \rfloor$ for each polling task and do not use the suggested binary search from [20] to reduce runtime, as described in [24]. We note, however, that a more brute-force search for every T^p up to and beyond the hyperperiod may improve schedulability but will also lead to a significant runtime increase[1]. 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 [58], favoring AdvPoll even more over our approach in terms of runtime. Similar to [24], we use an LLF schedule simulation until the hyperperiod for both SPoll and AdvPoll to generate the static TT schedule with the found polling task(s). We also compare to Slot-Shifting [45] by implementing the offline schedulability test for ET tasks and using an LLF simulation to generate the TT schedule, as described in [24]. We note that it is difficult to compare to Slot-Shifting fairly since the schedulability of Slot-Shifting is highly dependent on the heuristic to initially create and then change the TT schedule. However, the authors of [45], [44], [30] do not provide an algorithm to guide the TT schedule changes in case of ET infeasibility. If the ET tasks are not schedulable with the initial TT schedule, we generate a new LLF-based schedule using both TT and ET tasks as periodic tasks, mark the ET slots as empty, and try again. For arbitrary deadlines, which AdvPoll and Slot-Shifting do not support, we have taken the simplification of computing schedulability using $min(D_i, T_i)$.

All algorithms were implemented in Python, and all experiments were run on an Apple MacBook M1 Pro 10-core (3.12GHz) with 16GB RAM, using only 1 core per test case.

A. Synthetic test cases

We extended the task generator from [59], [60] to create task sets with TT and ET tasks with a deadline-monotonic priority assignment. All generated task sets are schedulable if the ET tasks are considered periodic TT tasks.

For the first set of experiments, we compare the approaches in terms of schedulability and runtime for use cases with different period sets and microtick values. For **test suite 1** (Figs. 6 and 8), we selected a microtick of $250\mu s$ and periods selected from the set $\{5, 10, 20, 40, 80\}ms$ with a distribution of $\{9.166\%, 26.66\%, 12.5\%, 19.166\%, 32.5\%\}$. For the period values and distribution of this test suite, we analyzed a real-world automotive use case (c.f. Sec. IV-B) containing 120 tasks, generating similar task sets. For **test suite 2** (Figs. 7 and 9), we selected a $100\mu s$ microtick and used periods which correspond to those found in automotive applications [61], [62], [27], namely $\{1, 2, 5, 10, 20, 50, 100, 200, 1000\}ms$. We try to emulate the realistic period distribution described in Table III of [62], without the angle-synchronous activation. We also performed experiments using a microtick of $1ms$ and periods uniformly chosen from the set $\{200, 300, 400\}ms$ (**test suite 3**) and with a $10\mu s$ microtick and periods of $\{20, 30, 40\}ms$ (**test suite 4**) which confirmed our results but are not included due to space limitations. In practice, a $10\mu s$ microtick will infer too much overhead. For real-world overhead measurements of different microticks in embedded TT systems, we refer the reader to Fig. 6 of [55] and note that our method can be applied to any microtick.

For all synthetic test sets, we use 30 TT and 20 ET tasks per task set and 100 task sets per test case, similar to [24]. For the constrained ET deadline test cases, we select D_i uniformly in the upper half of the interval $[C_i, T_i]$, and for arbitrary ET deadlines, we choose D_i uniformly in the interval $[C_i, 5 \cdot T_i]$. We vary $U^{TT}, U^{ET} \in \{0.1, 0.2, .., 0.7\}$ such that $U^{TT} + U^{ET} \leq 0.9$ resulting in 34 tuples (U^{TT}, U^{ET}) as seen on the x-axis of Figs. 6 - 9. In some cases, there is a minor difference in terms of schedulability between LMminB3LF and B3LF $(1 - 4\%)$ since we consider only multiples of I^{TT} for the values of L_M in LMminB3LF. However, since, in most cases, the schedulability is the same, we do not show them individually in the plots to maintain readability.

For constrained-deadline systems (Fig. 6 and 7), our method consistently outperforms AdvPoll, SPoll, and Slot-shifting in terms of schedulability (left y-axis) for all tested systems, especially as the TT task utilization increases. All other methods

[1]We ran the brute-force AdvPoll with $(U^{TT}, U^{ET}) = (20, 40)$ and $(30, 50)$ from Fig. 7. The AdvPoll schedulability increased from 50% to 57% and from 30% to 49%. The avg. runtime increased from 4.2s to 2.1min and from 6.9s to 3.1min. Our B3LF method was still superior with 65% and 52% schedulability and avg. runtime of 144ms and 214ms, respectively.

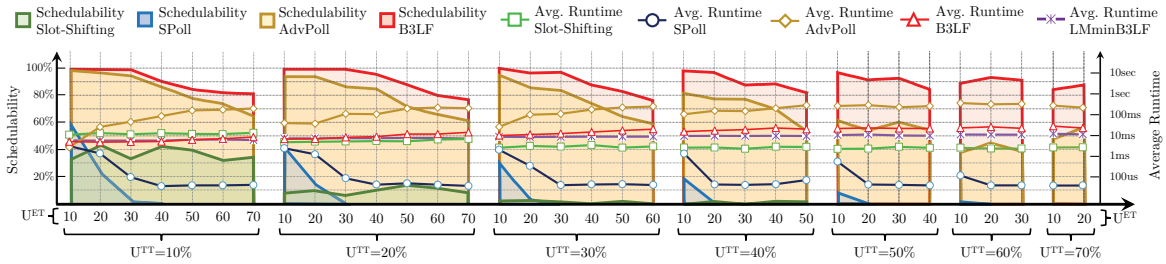


Fig. 6: Schedulability and avg. runtime with $250\mu s$ microtick, periods $T_i \in \{5, 10, 20, 40, 80\}ms$ and constrained ET deadlines in $[C_i, T_i]$.

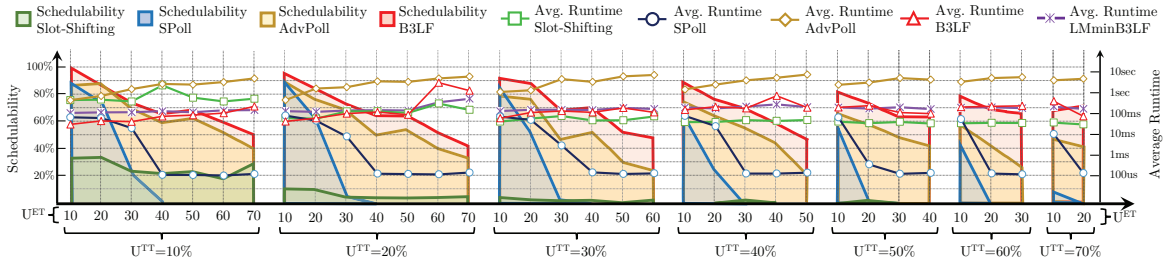


Fig. 7: Schedulability and avg. runtime with $100\mu s$ microtick, periods $T_i \in \{1, 2, 5, 10, 20, 50, 100, 200, 1000\}ms$ and constrained ET deadlines $\in [C_i, T_i]$.

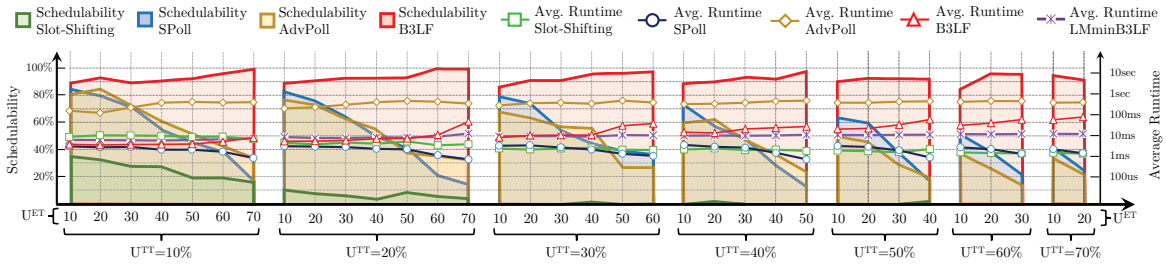


Fig. 8: Schedulability and avg. runtime with $250\mu s$ microtick, periods $T_i \in \{5, 10, 20, 40, 80\}ms$ and arbitrary ET deadlines in $[C_i, 5 \cdot T_i]$.

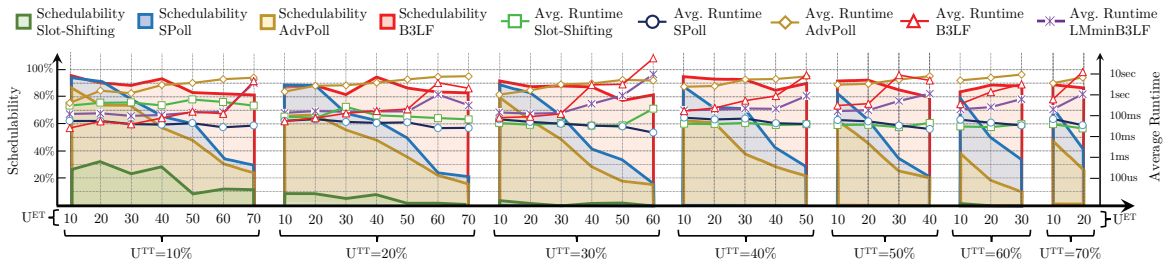
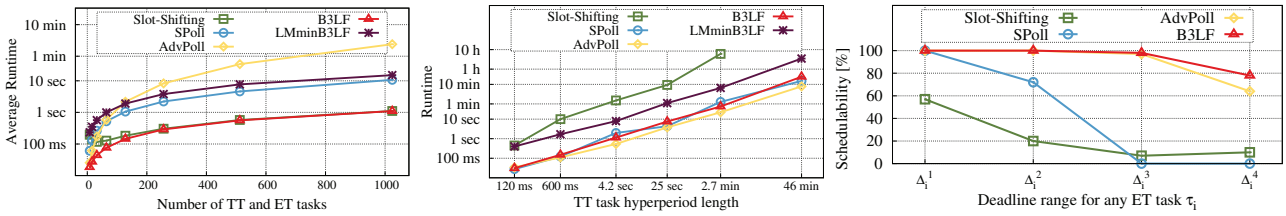


Fig. 9: Schedulability and avg. runtime with $100\mu s$ microtick, periods $T_i \in \{1, 2, 5, 10, 20, 50, 100, 200, 1000\}ms$ and arbitrary ET deadlines in $[C_i, 5 \cdot T_i]$.

fail to reach comparable schedulability except for AdvPoll when the system utilization is low. For arbitrary deadlines (Figs. 8 and 9), we achieve a higher average schedulability rate (left y-axis) than all other methods in almost all cases, often by a significant amount (especially in highly utilized systems). In some isolated cases, SPoll is better by $1-2\%$ for $(U^{TT}, U^{ET}) = (10\%, 20\%), (20\%, 10\%)$ in Fig. 9. We note that harmonic periods are common in real-world systems: e.g., the AFDX aerospace standard defines periods as powers of 2 multiplied by 1ms between $1-128$ms [63]. However, we also include highly non-harmonic periods leading to a hyperperiod explosion in Fig. 10b, showing the runtime comparison of the methods. We also change randomly some periods from $(10ms, 20ms, 40ms)$ to $(10.25ms, 20.75ms, 40.25ms)$ in use-cases $(U^{TT}, U^{ET}) = (10\%, 70\%), (70\%, 10\%), (70\%, 20\%)$ from **test suite 1** (Fig. 6). The schedulability results are $(12\%, 0\%, 0\%)$ for Slot-shifting, $(0\%, 0\%, 0\%)$ for SPoll, $(40\%, 40\%, 20\%)$ for AdvPoll, and $(88\%, 76\%, 92\%)$ for B3LF, confirm our previous schedulability results.

In terms of runtime (right logarithmic y-axis of Figs. 6, 9), our method is faster than Slot-Shifting, SPoll, and AdvPoll in most but not all cases. SPoll is faster when it cannot find feasible schedules (since the algorithm ends at the first polling task for which the oversampling leads to infeasibility) but



(a) increasing number of tasks (b) increasing TT hyperperiod (c) decreasing event-triggered task laxity

Fig. 10: Complexity and schedulability experiments for different problem dimensions.

cannot compete in terms of schedulability. Similarly, Slot-Shifting is sometimes faster due to the efficient schedulability test for ET tasks or when it cannot find a schedule, but it also cannot compete in terms of schedulability. AdvPoll is only comparable in runtime with B3LF for poorly utilized systems where the greedy search for the server period succeeds early. In cases where classical methods have a comparable schedulability rate to our method, B3LF almost always achieves a faster average runtime for the schedule generation. The comparison of B3LF to the LMminB3LF design optimization in terms of runtime yields a maybe counterintuitive result. If there are very small and very large periods (test suites 1 & 2), B3LF is sometimes slower than LMminB3LF, meaning that for some test cases, computing the TT schedule multiple times and calculating the ET delays once, as is done in LMminB3LF, takes less time than calculating the ET delays multiple times and computing the TT schedule once, as is done in B3LF. For the latter case, calculating the precise ET delay is more time-consuming due to the large number of linear pieces needed to describe the staircase functions, and a more significant timeline duration needs to be explored to find the precise delays for highly loaded systems.

Holistic scheduling [18], [19], [41] assumes that ET tasks have periodic activation (with bounded jitter) and uses a specific ET schedulability that is not suitable for our more generic sporadic ET task model. Hence, we cannot directly compare it to our method. However, we use the ALAP/ASAP-based heuristic from [18], [19], [41] with the changed simulation-based schedulability test for sporadic tasks proposed in [24]. We ran the constrained-deadline experiments with test suite 1 and the schedulability of holistic scheduling was roughly in between SPoll and AdvPoll (c.f. Fig. 6) and much lower than the schedulability of our method in all cases except $(U^{TT}, U^{ET}) = (10\%, 40\%), (10\%, 50\%)$. The avg. runtime for holistic scheduling was 2404ms compared to 17ms for our method. We do not show the complete results since, as described above, the original holistic method cannot be applied to our more general sporadic model.

In the second set of experiments, we study the runtime of our approach for increasing number of tasks (Fig. 10a), (rapidly) increasing hyperperiods (Fig. 10b), and decreasing laxity (Fig. 10c). We use a test setup with a very small microtick ($10\mu s$) in order to stress our algorithm when generating the schedule table. In Fig. 10a, we generate 100 constrained-deadline task sets per test case with each having 20% ET and 20% TT task utilization, and periods chosen from the set $\{50, 100\}ms$. 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 LMminB3LF, showing the efficiency of our method in relation to SPoll and AdvPoll. We see that the runtime of all methods grows linearly with the number of tasks and that B3LF and Slot-Shifting have the lowest overall average runtime out of all methods. In Fig. 10b, we generate 1 implicit deadline task set per test case with 8 TT and 8 ET tasks, increasing the hyperperiod of TT tasks (HP) 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 while the x-axis shows the TT hyperperiod HP, the schedule cycle (the size of the static schedule table) is a function of HP, being either equal to it or a multiple thereof. For SPoll and AdvPoll, the schedule cycle is the lcm between HP and the period(s) of the polling task(s), and for B3LF and LMminB3LF, it may be a multiple of HP (c.f. Alg. 1) upper bounded by $\lfloor \frac{L_M}{HP} \rfloor HP$. As an example, in our test with a TT hyperperiod of $HP = 46min$, the schedule cycle for AdvPoll and B3LF equals HP, and for SPoll it is $\approx 1.5h$. Slot-shifting reaches our set timeout of 10 hours for $HP = 46min$ since it is more susceptible to increasing schedule sizes due to testing all critical slots until HP and the bookkeeping of the reserved slot list. We note that even for the case with an unrealistically large schedule cycle of $\approx 46min$ and an unrealistically small microtick of $10\mu s$, B3LF manages to compute a schedule table in $26min$, which is quite acceptable for an offline schedule generation tool. Moreover, we note that in B3LF, the computation of the maximum burst is independent of HP. Finally, we study the effect of task laxity on schedulability for each method. In Fig. 10c, we generate 100 constrained-deadline task sets, each having 4 TT and 4 ET tasks with 40% and 20% system utilization, respectively. For each ET task $\tau_i \in \mathcal{T}^{ET}$, we choose the deadline randomly in each quartile of the interval $[C_i, T_i]$ in decreasing order (x-axis), i.e., $\Delta_i^k = [T_i - k(T_i - C_i)/4, T_i - (k-1)(T_i - C_i)/4], k = 1, ..., 4$, leading to an increasingly smaller laxity. While for Δ_i^1, the schedulability of most methods is at 100%, Slot-Shifting and

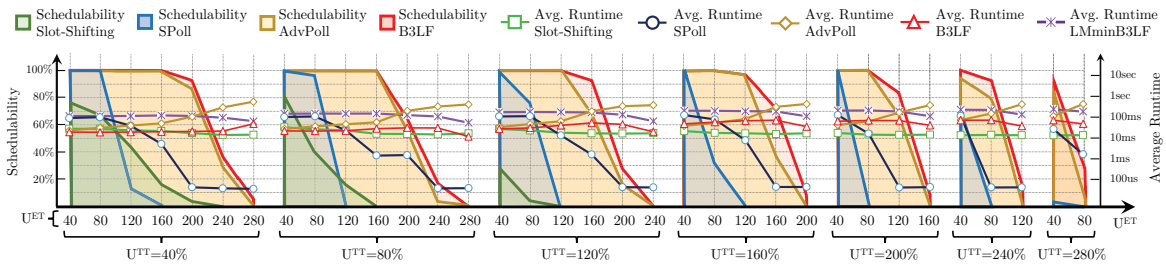


Fig. 11: Schedulability and avg. runtime on a 4-core platform with $250\mu s$ microtick, periods $T_i \in \{5, 10, 20, 40, 80\}ms$ and constrained ET deadlines.

SPoll have low schedulability as the laxity decreases. AdvPoll has slightly lower schedulability compared to our method for Δ_i^4 with 64% vs. 78% schedulability.

The size of the schedule tables depends on the schedule cycle and the microtick. We computed the average memory sizes of the generated TT schedules for all 4 test suites from Sec. IV-A. The schedules produced with B3LF have relatively small sizes (comparable to the other methods) of 30KB to 500KB for realistic scenarios and 1MB for the unrealistic $10\mu s$ test suite 4, making them suitable for embedded targets.

B. Real-world test case

In the previous section, we have created for **test suite 1** a series of synthetic task sets based on the periods and period distributions of a real-world automotive system currently on the road in millions of vehicles. We cannot reveal any sensitive information or intellectual property due to confidentiality but note that the application features a multi-SoC multi-core platform with 6 cores dedicated to critical applications. There are 120 tasks with periods in the set $\{5, 10, 20, 40, 80\}ms$ and a distribution of $\{9.166\%, 26.66\%, 12.5\%, 19.166\%, 32.5\%\}$ and a total utilization of 3.08. In the original application, all the tasks are defined as TT and are pre-assigned to the 6 dedicated cores. Hence, in this experimental evaluation, we run the single-core implementation for each core with the assigned tasks. Since all tasks were initially defined as TT and there were no ET tasks, we change the type of the task to ET if the task laxity (i.e., $T_i - C_i$) is larger than 100 microticks. In total, 73 tasks were converted from TT to ET. For these ET tasks, we set the priority of a task τ_i to $p(i) = max(0, 6 - \lfloor (T_i - C_i)/100 \rfloor)$. All tasks with laxity smaller than or equal to 100 microticks remain TT tasks. All methods manage to successfully compute schedules for all 6 cores, except for SPoll, which only finds feasible schedules for 4 cores. The average runtimes are as follows: 0.56ms for B3LF, 2ms for LMminB3LF, 1.35ms for SPoll, 24.9ms for AdvPoll, and 6.7ms for Slot-Shifting. The real-world test case confirms our synthetic benchmark results, with B3LF being faster than all other methods while having equal or better schedulability.

C. Multicore test cases and dependencies

We show the applicability of our method to partitioned multi-core systems using the automotive configuration from **test suite 1** and our simple allocation heuristic from Sec. III-E. We merge the original 100 input task sets in sets of 4 to create test cases for a multi-core platform with 4 cores, resulting in 25 task sets per input configuration. We only show the constrained deadline test cases since the main objective of the evaluation is to show the applicability of our method, not the effectiveness of the allocation heuristic. In Fig. 11, we depict, as before, the schedulability on the left y-axis and the runtime on the logarithmic y-axis for different (U^{TT}, U^{ET}) configurations. In most cases, we see that B3LF has better schedulability than all other methods and is almost always faster than the only comparable method in terms of schedulability (AdvPoll).

Finally, we use the real-world example from Sec. IV-B, with the same task type change and priority assignment as described before. However, we assume that tasks are not pre-assigned to cores and use our simple heuristic detailed in Sec. III-E to do the assignment with an increasing number of available cores. With 4 cores, only SPoll fails to produce schedules, while B3LF, LMminB3LF, AdvPoll, and Slot-Shifting succeed with the following runtimes: 2.59ms, 17.87ms, 312.49ms, and 55.89ms, respectively. SPoll only manages to find schedules when we increase the number of cores to 17.

V. Conclusion

We have studied the integration of sporadic event-triggered (ET) tasks with arbitrary deadlines into static time-triggered (TT) schedules. Our novel method distills a burst limiting constraint (BLC) for TT tasks that guarantees the schedulability of ET tasks and that can be integrated into any TT schedule generation algorithm. We also introduced our own TT schedule generation method that respects the BLC and thereby fulfills the temporal requirements of both TT and ET tasks. We have also presented a design optimization technique for iterative design processes where ET tasks are added/changed later and an allocation heuristic for applying our method to partitioned multi-core systems. We have shown through an extensive series of synthetic and real-world test cases that our method outperforms state-of-the-art approaches in terms of schedulability and runtime. The BLC derived with our method is independent of our proposed LLF-based synthesis algorithm and represents an independent and generic constraint on TT slot placement that can be readily integrated into any existing schedule generation method for time-triggered systems, even those that include complex multi-rate cause-effect chains and network communication dependencies.

ACKNOWLEDGMENTS

This Project is under Grant Agreement Preparation by the Chips-JU for funding in the frame of the Call 2023 activities.

REFERENCES

[1] T. Fleming, S. K. Baruah, and A. Burns, "Improving the schedulability of mixed criticality cyclic executives via limited task splitting," in *Proc. RTNS*, 2016.

[2] S. K. Baruah and G. Fohler, "Certification-cognizant time-triggered scheduling of mixed-criticality systems," in *Proc. RTSS*, 2011, pp. 3–12.

[3] B. Akesson, M. Nasri, G. Nelissen, S. Altmeyer, and R. I. Davis, "A comprehensive survey of industry practice in real-time systems," *Real-Time Systems*, vol. 58, no. 3, pp. 358–398, Sep 2022.

[4] M. Lukasiewycz, R. Schneider, D. Goswami, and S. Chakraborty, "Modular scheduling of distributed heterogeneous time-triggered automotive systems," in *Proc. ASP-DAC*, 2012.

[5] F. Sagstetter, S. Andalam, P. Waszecki, M. Lukasiewycz, H. Stähle, S. Chakraborty, and A. Knoll, "Schedule integration framework for time-triggered automotive architectures," in *Proc. DAC*, 2014.

[6] R. Ernst, S. Kuntz, S. Quinton, and M. Simons, "The Logical Execution Time Paradigm: New Perspectives for Multicore Systems (Dagstuhl Seminar 18092)," *Dagstuhl Reports*, vol. 8, no. 2, pp. 122–149, 2018.

[7] G. Niedrist, "Deterministic architecture and middleware for domain control units and simplified integration process applied to ADAS," in *Fahrerassistenzsysteme 2016*. Springer Fachmedien Wiesbaden, 2018.

[8] T. Fleming and A. Burns, "Investigating mixed criticality cyclic executive schedule generation," in *Proc. WMC*, 2015.

[9] S. D. McLean, E. A. Juul Hansen, P. Pop, and S. S. Craciunas, "Configuring ADAS platforms for automotive applications using metaheuristics," *Frontiers in Robotics and AI*, vol. 8, p. 353, 2022.

[10] P. Karachatzis, J. Ruh, and S. S. Craciunas, "An evaluation of time-triggered scheduling in the linux kernel," in *Proc. RTNS*. ACM, 2023.

[11] M. Becker, D. Dasari, S. Mubeen, M. Behnam, and T. Nolte, "Synthesizing job-level dependencies for automotive multi-rate effect chains," in *Proc. RTCSA*, 2016.

[12] M. Becker, D. Dasari, S. Mubeen, M. Behnam, and T. Nolte, "End-to-end timing analysis of cause-effect chains in automotive embedded systems," *Journal of Systems Architecture*, vol. 80, 2017.

[13] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga, "Time-triggered mixed-critical scheduler on single and multi-processor platforms," in *Proc. HPCC*, 2015, pp. 684–687.

[14] J. Real, S. Sáez, and A. Crespo, "A hierarchical architecture for time- and event-triggered real-time systems," *J. Syst. Archit.*, vol. 101, 2019.

[15] J. Xu and D. L. Parnas, "Priority scheduling versus pre-run-time scheduling," *Real-Time Syst.*, vol. 18, no. 1, p. 7–23, 2000.

[16] C. D. Locke, "Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives," *Real-Time Syst.*, vol. 4, no. 1, p. 37–53, 1992.

[17] A. Minaeva and Z. Hanzálek, "Survey on periodic scheduling for time-triggered hard real-time systems," *ACM Comput. Surv.*, vol. 54, no. 1, 2021.

[18] T. Pop, P. Eles, and Z. Peng, "Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems," in *Proc. CODES*. ACM, 2002.

[19] T. Pop, P. Eles, and Z. Peng, "Schedulability analysis for distributed heterogeneous time/event triggered real-time systems," in *Proc. ECRTS*, 2003.

[20] L. Almeida and P. Pedreiras, "Scheduling within temporal partitions: Response-time analysis and server design," in *Proc. EMSOFT*, 2004.

[21] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *Proc. RTSS*. IEEE, 2003.

[22] I. Shin and I. Lee, "Compositional real-time scheduling framework with periodic model," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, 2008.

[23] S. Saewong, R. R. Rajkumar, J. P. Lehoczky, and M. H. Klein, "Analysis of hierarchical fixed-priority scheduling," in *Proc. ECRTS*, 2002.

[24] C. Meroni, S. S. Craciunas, A. Finzi, and P. Pop, "Mapping and integration of event- and time-triggered real-time tasks on partitioned multi-core systems," in *Proc. ETFA*. IEEE, 2023.

[25] P. Muoka, D. Onwuchekwa, and R. Obermaisser, "Adaptive scheduling for time-triggered network-on-chip-based multi-core architecture using genetic algorithm," *Electronics*, vol. 11, no. 1, 2022.

[26] C. Deutschbein, T. Fleming, A. Burns, and S. K. Baruah, "Multi-core cyclic executives for safety-critical systems," *Science of Computer Programming*, vol. 172, pp. 102–116, 2019.

[27] A. Hamann, D. Dasari, S. Kramer, M. Pressler, and F. Wurst, "Communication Centric Design in Complex Automotive Embedded Systems," in *Proc. ECRTS*, 2017.

[28] S. D. McLean, S. S. Craciunas, E. A. Juul Hansen, and P. Pop, "Mapping and scheduling automotive applications on ADAS platforms using metaheuristics," in *Proc. ETFA*. IEEE, 2020.

[29] H. Kopetz, "Sparse time versus dense time in distributed real-time systems," in *Proc. ICDCS*, 1992.

[30] D. Isović and G. Fohler, "Handling mixed sets of tasks in combined offline and online scheduled real-time systems," *Real-Time Syst.*, vol. 43, no. 3, 2009.

[31] H. Aydin, R. Melhem, D. Mossé, and P. Mejia-Alvarez, "Power-aware scheduling for periodic real-time tasks," *IEEE Transactions on Computers*, vol. 53, no. 5, pp. 584–600, 2004.

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

[33] L. Thiele, S. Chakraborty, and M. Naedele, "Real-time calculus for scheduling hard real-time systems," in *Proc. ISCAS*, vol. 4, 2000.

[34] D. V. R. Sudhakar, K. Albers, and F. Slomka, "Generalized and scalable offset-based response time analysis of fixed priority systems," *Journal of Systems Architecture*, vol. 112, p. 101856, 2021.

[35] M. Moy and K. Altisen, "Arrival curves for real-time calculus: the causality problem and its solutions," in *Proc. TACAS*, 2010.

[36] S. Chakraborty, S. Künzli, and L. Thiele, "A general framework for analysing system properties in platform-based embedded system designs," in *Proc. DATE*, 2003.

[37] Y. Tang, Y. Jiang, X. Jiang, and N. Guan, "Pay-burst-only-once in real-time calculus," in *Proc. RTCSA*, 2019.

[38] E. Wandeler and L. Thiele, "Real-time interfaces for interface-based design of real-time systems with fixed priority scheduling," in *Proc. EMSOFT*, 2005.

[39] A. Bouillard, M. Boyer, and E. Le Corronc, *Deterministic Network Calculus – From theory to practical implementation*. Wiley, 2018.

[40] M. Boyer, P. Roux, H. Daigmorte, and D. Puechmaille, "A residual service curve of rate-latency server used by sporadic flows computable in quadratic time for network calculus," in *Proc. ECRTS*, 2021.

[41] T. Pop, "Scheduling and optimisation of heterogeneous time/event-triggered distributed embedded systems," Ph.D. dissertation, Linköping University, 2003.

[42] A. Zabos, "Temporal partitioning of flexible real-time systems," Ph.D. dissertation, University of York, 2011.

[43] G. Fohler, "Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems," in *Proc. RTSS*, 1995.

[44] D. Isović and G. Fohler, "Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints," in *Proc. RTSS*, 2000.

[45] D. Isović and G. Fohler, "Handling sporadic tasks in off-line scheduled distributed real-time systems," in *Proc. ECRTS*, 1999, pp. 60–67.

[46] P. Kumar and L. Thiele, "Cool shapers: Shaping real-time tasks for improved thermal guarantees," in *Proc. DAC*, 2011.

[47] S. Thangamuthu, N. Concer, P. J. L. Cuijpers, and J. J. Lukkien, "Analysis of ethernet-switch traffic shapers for in-vehicle networking applications," in *Proc. DATE*, 2015.

[48] J. Imtiaz, J. Jasperneite, and L. Han, "A performance study of ethernet audio video bridging (AVB) for industrial real-time communication," in *Proc. ETFA*. IEEE, 2009.

[49] A. Finzi, A. Mifdaoui, F. Frances, and E. Lochin, "Incorporating TSN/BLS in AFDX for mixed-criticality applications: Model and timing analysis," in *Proc. WFCS*, 2018.

[50] A. Finzi and A. Mifdaoui, "Worst-case timing analysis of AFDX networks with multiple TSN/BLS shapers," *IEEE Access*, vol. 8, 2020.

[51] F.-J. 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 19.09.2023.

[52] S. S. Craciunas, R. Serna Oliver, M. Chmelik, and W. Steiner, "Scheduling real-time communication in IEEE 802.1Qbv Time Sensitive Networks," in *Proc. RTNS*. ACM, 2016.

[53] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, 1973.

[54] J. Leung, "A new algorithm for scheduling periodic, real-time tasks," *Algorithmica*, vol. 4, no. 1, pp. 209–219, 1989.

[55] S. S. Craciunas, R. Serna Oliver, and V. Ecker, "Optimal static scheduling of real-time tasks on distributed time-triggered networked systems," in *Proc. ETFA*. IEEE, 2014.
[56] J. Hildebrandt, F. Golatowski, and D. Timmermann, "Scheduling coprocessor for enhanced least-laxity-first scheduling in hard real-time systems," in *Proc. ECRTS*, 1999.
[57] B. B. Brandenburg and M. Gül, "Global scheduling not required: Simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations," in *Proc. RTSS*, 2016, pp. 99–110.
[58] S. K. Baruah, L. E. Rosier, and R. R. Howell, "Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor," *Real-Time Syst.*, vol. 2, no. 4, 1990.
[59] P. Emberson, R. Stafford, and R. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *Proc. WATERS*, 2010.
[60] P. Emberson, R. Stafford, and R. Davis, "A taskset generator for experiments with real-time task sets," available at https://github.com/jlelli/taskgen, Accessed on 19.09.2023.
[61] S. Tobuschat, R. Ernst, A. Hamann, and D. Ziegenbein, "System-level timing feasibility test for cyber-physical automotive systems," in *Proc. SIES*, 2016.
[62] S. Kramer, D. Ziegenbein, and A. Hamann, "Real world automotive benchmarks for free," in *Proc. WATERS*, 2015.
[63] M. Li, M. Lauer, G. Zhu, and Y. Savaria, "Determinism enhancement of AFDX networks via frame insertion and sub-virtual link aggregation," *IEEE Transactions on Industrial Informatics*, vol. 10, no. 3, pp. 1684–1695, 2014.