Real-Time Systems (2022) 58:313–357
https://doi.org/10.1007/s11241-022-09385-8

Robust and accurate regression-based techniques for period inference in real-time systems

Şerban Vădineanu[1] · **Mitra Nasri**[2]

Accepted: 3 May 2022 / Published online: 20 June 2022
© The Author(s) 2022

Abstract

With the growth in complexity of real-time embedded systems, there is an increasing need for tools and techniques to understand and compare the observed runtime behavior of a system with the expected one. Since many real-time applications require periodic interactions with the environment, one of the fundamental problems in guaranteeing their temporal correctness is to be able to infer the periodicity of certain events in the system. The practicability of a period inference tool, however, depends on both its accuracy and robustness (also its resilience) against noise in the output trace of the system, e.g., when the system trace is impacted by the presence of aperiodic tasks, release jitters, and runtime variations in the execution time of the tasks. This work (i) presents the first period inference framework that uses regression-based machine-learning (RBML) methods, and (ii) thoroughly investigates the accuracy and robustness of different families of RBML methods in the presence of uncertainties in the system parameters. We show, on both synthetically generated traces and traces from actual systems, that our solutions can reduce the error of period estimation by two to three orders of magnitudes w.r.t. the state of the art.

Keywords Period inference · Regression-based machine learning · Robust learning · Real-time systems · Event traces

✉ Mitra Nasri
m.nasri@tue.nl

Şerban Vădineanu
s.vadineanu@liacs.leidenuniv.nl

[1] Leiden University, Leiden, Netherlands

[2] Eindhoven University of Technology, Eindhoven, Netherlands

1 Extended version

This paper builds upon and extends the preliminary paper "Robust and Accurate Period Inference using Regression-Based Techniques" (Vădineanu and Nasri 2020) by:

(i) Introducing a more informative type of input data, called *quaternary projection* (Sect. 2.2), that includes the intervals during which 'lower-priority tasks' were occupying the resource. We use this information to derive yet a tighter bound on the period (see Sect. 4.2 for the derivation of the bound and Sect. 5.7 for a comparison of the bounds derived from ternary and quaternary projections);
(ii) Tightening the existing upper bounds derived from ternary projections w.r.t. Vădineanu and Nasri (2020) (see Sect. 4.1.2 and a comparison between the old and the new bound in Sect. 5.8);
(iii) Introducing a new upper bound for ternary and quaternary projections that copes with release jitter of the tasks (see Sect. 4.1.3);
(iv) Adding extensive experiments to further evaluate the *robustness* of our solution when it is applied on (i) different *scheduling policies* (see Sect. 5.6.5), (ii) overloaded systems in which the total utilization is larger than 1 (see Sect. 5.4.3), and (iii) tasks that have arbitrary offsets (see Sects. 5.4.4 and 5.9);
(v) An extensive experiment to evaluate our solution when it is used for non-preemptive task sets. We studied the impact of the number of tasks, utilization, execution time variation, release jitter, and offsets on the performance of our solution. We showed that our method still has superior accuracy in comparison with the state of the art even for simpler problems such as non-preemptive tasks (see Sect. 5.9);
(vi) Adding more examples and discussions to the space-pruning method (see Sect. 4) section and some discussions on how to obtain projections from a system in practice (see Sect. 2.3).

2 Introduction

The rapid growth of software size and complexity in real-time embedded systems has posed imminent challenges to the ability to debug systems, identify runtime deviations from the correct service (Young et al. 2019), and detect (and evade) security attacks at runtime (Nasri et al. 2019). This raises an urge for tools and techniques to understand (or infer) the runtime behavior of a system from its observable outputs such as the traces of output messages, task executions, actuations, etc. without impacting the system itself or, in some cases, without being able to access the source code or the internal parts of the system.

In this paper, we focus on developing a tool for inferring the timing properties of a system. Such a tool can be used to (i) find time-bugs during the development phase, for example, to check if activities happen with the expected frequency or period, or to act as an automated test oracle (Barr et al. 2015), (ii) detect timing

anomalies and security attacks that leave a trace on the observable timing profile of the system during the operation phase (e.g., such as those explained by Nasri et al. (2019), Salem et al. (2016), and Iegorov and Fischmeister (2018) to spot anomalies in the regularity of an activity in the system), and (iii) diagnosing the system after applying a patch or an upgrade during the maintenance phase (e.g., to check if a data-consumer application still performs periodically after installing an upgrade on the data-producer application).

Since many real-time applications require periodic interactions with the environment (Akesson et al. 2020), one of the primary use cases of a timing inference tool is to infer the periodicity of events from a system's output traces (Berberidis et al. 2002; McKilliam et al. 2014; Puech et al. 2019). What makes this very first step challenging is that the observable timing traces are typically obtained from the components' interfaces and hence are impacted by the internal structure of the application, operating system, hardware platform, and their interactions. For instance, consider an execution trace that indicates the time intervals during which a certain task has occupied the processor. It is easy to infer the period if the task exclusively runs on top of dedicated hardware. It becomes harder if the task is one of the low-priority tasks in a set of periodic tasks running on top of a *real-time operating system* (RTOS) with a preemptive fixed-priority scheduling (FP) policy because then the task's execution intervals are affected (e.g., preempted) due to the interference from the higher-priority periodic tasks. Finally, it becomes much harder if the latter system also includes high-priority aperiodic or event-driven activities (such as interrupt services), sporadic tasks, release jitters, and deadline misses. A timing inference tool, therefore, must be *robust* against these interferences, dynamic behavior, and uncertainties; otherwise, it might not be able to address true challenges faced by real systems and hence becomes useless in practice. Furthermore, it must be accurate, else it will not be helpful to find time bugs or to detect deviations from the expected periodicity.

Related work. Iegorov et al. (2017) are among the few pioneers who proposed a solution for the problem of inferring periods from execution traces. They created an algorithm which identifies the time intervals between consecutive jobs and computed the period as the mode of the intervals' distribution. However, their method performs poorly when the tasks have runtime execution-time variation and/or the true period of the task under analysis does not divide all other smaller periods in the task set, i.e., it is not *harmonic* with the rest of the tasks. Young et al. (2019) use a *fast Fourier transformation* to infer the periodicity of messages sent on a *controller area network* (CAN) in order to detect security attacks that impact the timing of the messages. Their problem, however, is only a subset of ours since CAN applies a non-preemptive fixed-priority policy and messages have typically a fixed size with a low runtime variation on the message length.

Data-driven methods such as *k*-nearest neighbors and dynamic time-warping algorithms as well as *long short-term memory* (LSTM) neural networks have been used in reverse engineering real-time systems to identify tasks from their runtime power traces by Lamichhane et al. (2018) and to reconstruct traces affected by noise by Sucholutsky et al. (2019). However, to the best of our knowledge, no study so far has utilized *regression-based machine learning* (RBML) methods to infer the timing

properties of real-time systems. We not only provide the first such solution, but also extensively investigate the accuracy and robustness of various families of RBML for this problem.

Finding the periodicity of a signal is a well-studied problem in signal processing research (Schuster 1898; Berberidis et al. 2002; Vlachos et al. 2005; Li 2012; McKilliam et al. 2014; Malode et al. 2015; Unnikrishnan and Jothiprakash 2018; Puech et al. 2019; Gubner 2006). *Periodogram* (Schuster 1898) and *circular autocorrelation* (Gubner 2006) are among the widely used methods to find a plausible set of periods for a signal. However, as we will see in the experimental section, these methods perform poorly when used on signals generated from preempted tasks. Nonetheless, despite their limitations, we found them to be helpful to generate an initial set of candidate periods and hence will use them only in the first step of our solution to extract features from execution traces.

This paper. We consider the problem of inferring a task's period from a timed-sequence of zeros and ones (called a *binary projection*) that shows when the task was occupying the resource (see Sect. 2). We consider a single processing resource (it can be a CPU, a network link, a CAN bus, etc.) that is governed by a work-conserving *job-level fixed-priority* (JLFP) scheduling policy. We assume *no prior knowledge* about the number of other tasks in the system and their parameters, execution model (preemptive or non-preemptive), runtime execution-time variations, and release jitters.

Our framework uses two signal-processing techniques, i.e., periodogram and circular autocorrelation, to extract features from the binary projection, treat and reduce the size and the number of features, and then use them to train a set of RBML methods (in Sect. 3). This work (i) presents the first period inference framework[1] that utilizes RBML methods, and (ii) thoroughly investigates the accuracy and robustness of different families of RBML methods in the presence of uncertainties in the system parameters, noise resulted from aperiodic tasks in the input data or missed jobs.

Our results show that RBML methods infer tasks' periods with an average error of 0.4% (for periodic tasks with or without execution-time variation), 1.1% (for periodic tasks with release jitter), and 0.4% (for task sets with a mixture of periodic, sporadic, and aperiodic tasks) while the state of the art (Iegorov and Fischmeister 2018) has an average error of 1160%, 1950%, and 156%, respectively. On case studies from actual systems (Lee et al. 2017; Seo et al. 2018), the error of our (best) solution was below 1.7%. Sect. 6 provides insight on the strengths and weaknesses of different families of RBML methods for the problem of period inference.

[1] Available at: https://github.com/SerbanVadineanu/period_inference.

3 System model and problem definition

3.1 System model

We assume a system with a single (processing) resource (such as a CPU core, I/O or CAN bus, or a link on the network). The resource can be occupied/used by a set of tasks $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$, scheduled by a work-conserving *job-level fixed-priority* (JLFP) scheduling policy on the resource, i.e., only the highest-priority job among the ready jobs can be dispatched on the resource, where a job is an instance of a task in τ. JLFP policies include widely implemented/used scheduling algorithms in real-time systems such as the earlier-deadline first (EDF), fixed-priority (FP), and first-in-first-out (FIFO) scheduling policies. A work-conserving scheduling policy is the one that does not leave the resource idle if there is a task that is ready to occupy the resource. Furthermore, we assume no restriction on whether each task executes preemptively or non-preemptively.

A task in τ can be activated periodically, sporadically, or aperiodically. A periodic or sporadic task is identified by $\tau_i = (C_i^{min}, C_i^{max}, T_i, D_i, \sigma_i)$, where C_i^{min} and C_i^{max} are the *best-case* and *worst-case execution times* (BCET and WCET), T_i is the period, D_i is the relative deadline (which is assumed to be equal to the period), and σ_i is the maximum release jitter of the task. Following Audsley's convention (Audsley et al. 1993), we assume positive release jitter, i.e., the $k^{\text{th}} \geq 1$ job of a periodic task τ_i is supposed to be released during the interval $[(k-1)T_i, (k-1)T_i + \sigma_i]$ and its deadline is at $(k-1)T_i + D_i$.

If the task is sporadic, its period indicates the minimum-inter arrival time between its activations. An aperiodic task is identified by a 3-tuple $\tau_j = (C_j^{min}, C_j^{max}, D_j)$, where C_j^{min} and C_j^{max} are the BCET and WCET and D_j is the relative deadline of the task, respectively.

We further assume that all timing parameters are positive integer values in $\mathbb{N}^+$ with the exception of C_i^{min} and σ_i that can be 0. The total utilization of the system is denoted by U and is the sum of the utilization of all periodic and sporadic tasks, i.e., $U = \sum u_i$, where $u_i = C_i^{max}/T_i$. The *hyperperiod* of a task set, denoted by H, is the least common multiple of the periods.

A task τ_i generates an infinite number of instances, called jobs, during the lifetime of the system. We use $J_{i,j}$ to denote the j-th job of a task τ_i. The priority of a job $J_{i,j}$ is denoted by $p_{i,j}$ and is determined by the scheduling policy. We assume that at any time instant t, either one of the tasks in τ or the *idle task*, denoted by τ_0, is running on the resource.

A trace $\mathcal{T} = ([t_s, t_e], \langle \epsilon_1, \epsilon_2, \ldots, \epsilon_N \rangle)$ is a time-ordered sequence of symbols that represents a schedule generated by the JLFP scheduler for the task set $\tau \cup \{\tau_0\}$ from the time t_s to t_e. Each symbol ϵ_i in the trace $\mathcal{T}$ is an identifier (index) of a task that was occupying the resource at time i, where $i \in \{t_s, t_{s+1}, \ldots, t_e\}$. Hence, $\epsilon_i \in \{0, 1, \ldots, n\}$. The length of a trace is $|\mathcal{T}| = t_e - t_s$.

Figure 1a and b show a schedule of a task set with 4 tasks and the equivalent trace of that schedule.

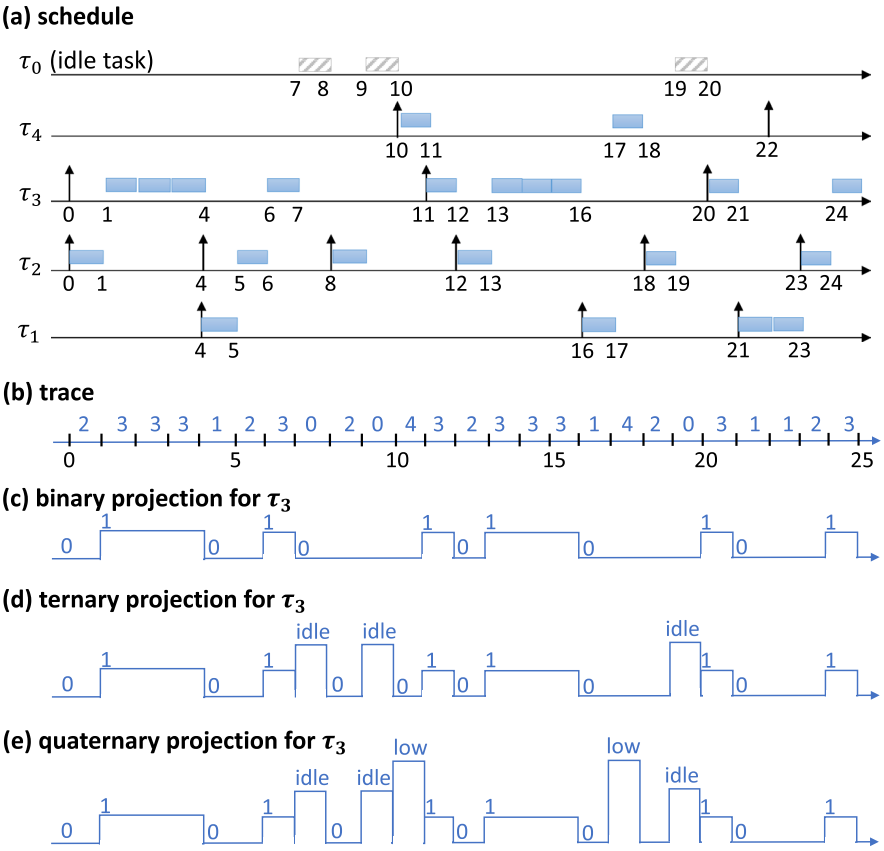

Fig. 1 A task set with one aperiodic task (τ_1 with $C_1^{max} = 2$), two sporadic tasks (τ_2 and τ_4 with $C_2^{max} = 1$ and $C_4^{max} = 2$) and one periodic task (τ_3 with $C_3^{max} = 4$ and $T_3 = 10$) with release jitter scheduled by a FP policy (assuming $p_i = i$). **a** Shows a schedule, **b** shows the trace of the schedule, and **c**–**e** show the binary, ternary, and quaternary projections of task τ_3 in the task set

3.2 Problem definition

To formally define the problems considered in the paper, we need to introduce three other notions that are tied to a trace: *binary projection*, *ternary projection*, and *quaternary projection* (shown in Fig. 1c to e).

A binary projection for a task τ_i is a sequence of zeros and ones that represents the times at which a job of the task under observation was occupying the resource in the trace. Figure 1c shows the binary projection of the task τ_3.

Definition 1 A *binary projection* of a trace $\mathcal{T}$ for a task τ_i, denoted by $P_i^B = \langle p_1, p_2, \ldots, p_{|\mathcal{T}|} \rangle$, is a time-ordered sequence of elements p_k, where

$$p_k = \begin{cases} 1 & \epsilon_k = i \\ 0 & \text{otherwise} \end{cases}. \tag{1}$$

A ternary projection (Fig. 1d) for a task contains resource-idle intervals in addition to what is stored in a binary projection.

Definition 2 A *ternary projection* of a trace $\mathcal{T}$ for a task τ_i, denoted by $P_i^T = \langle p_1, p_2, \ldots, p_{|\mathcal{T}|} \rangle$, is a time-ordered sequence of elements p_k, where

$$p_k = \begin{cases} 1, & \epsilon_k = i \\ idle, & \epsilon_k = 0 \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

A quaternary projection of a task τ_i is similar to the ternary projection except that it also includes the intervals in which a job of a lower-priority task than τ_i is occupying the resource. Namely, quaternary projections can be derived for FP scheduling policy (and not EDF).

Definition 3 A *quaternary projection* of a trace $\mathcal{T}$ for a task τ_i, denoted by $P_i^Q = \langle p_1, p_2, \ldots, p_{|\mathcal{T}|} \rangle$, is a time-ordered sequence of elements p_k, where

$$p_k = \begin{cases} 1, & \epsilon_k = i \\ idle, & \epsilon_k = 0 \\ low, & lp(\mathcal{T}, i, k, \epsilon_k) \\ 0, & \text{otherwise} \end{cases}, \tag{3}$$

where $lp(\mathcal{T}, i, k, \epsilon_k)$ returns *true* only if it is possible to verify that the job of τ_{ϵ_k} that occupies the resource at time k has a lower priority than the latest released job of τ_i at time k.

Figure 1e shows the quaternary projection of the task τ_3. As it can be seen, a low-priority task (τ_4) creates two time intervals, i.e., [10, 11) and [17, 18), with the label "low" in the quaternary projection of τ_3. Note that quaternary projections do not distinguish low-priority tasks from each other (namely, "low" can represent "any" of the low-priority tasks).

We conclude this section by defining three versions of the *period inference* (PI) problem:

Problem 1 Find the period of τ_i from its projection P_i^B.

Problem 2 Find the period of τ_i from its projection P_i^T provided that P_i^T does not include a deadline miss from τ_i.

Problem 3 Find the period of τ_i from its projection P_i^Q provided that P_i^Q does not include a deadline miss from τ_i.

It is worth noting that the only input to the Problems 1, 2, and 3 is a projection. Since a projection is just a sequence of limited symbols ('0', '1', 'idle', and 'low'), it does not contain any information about the scheduling policy or tasks' parameters (such as the execution times, release jitter, periods, etc.). Moreover, the projections themselves do not contain information about whether or not there are tardy jobs (i.e., jobs that have completed after their deadline) in the original trace.

3.3 Obtaining projections

In practice, one may utilize operating system commands such as *top* and *trace* commands or the Linux trace toolkit to obtain a trace of a certain task. *There is no need to distinguish or annotate preemptions, start of a new job, blocking times by lower-priority jobs, or self-suspensions* because that is the part that the period inference problem answers. It is also fine if the information gathered in a projection is incomplete (namely, misses some jobs of the task or some of its execution intervals).

In the context of a network resource, for example, on a CAN bus, one can obtain projections by observing the messages transferred on the bus to form a trace or to just use the outputs of the message filters of the CAN controller to get a binary, ternary, or quaternary projection for the message ID of interest.

Ternary and quaternary projections require slightly more detailed observations from the system. Still, the same tools mentioned above can be used to derive ternary projections (as the only added information in a ternary projection is the moments in which the resource is idle).

Quaternary projections are helpful only if there are "tasks" with a lower priority than the one being observed in the system (namely, the scheduling policy is not EDF). In our paper, we do not need to include information of *all* lower-priority tasks in a quaternary projection, i.e., only a partial observation would suffice (for example, only some of the lower-priority tasks can be observed but not all). The more information (about the lower-priority tasks) can be added to a quaternary projection, the better would be the period bounds that we will derive from the quaternary projections in Sect. 4.2.

In some cases, a task may roughly know the execution window of other higher- or lower-priority tasks if it collaborates with them, e.g., when it sends messages to them and waits until it receives an acknowledgement or response. However, when tasks are independent or isolated from other tasks, they may not be able to obtain information needed for quaternary projections without the help of an operating system. This may then restrict the applicability of quaternary projections to cases where the operating system also takes part in the safety-monitoring activity. For example, consider a case where a system is equipped with a safety-monitoring component whose goal is to ensure that certain activities happen periodically within an expected period range. To improve monitoring accuracy, the architect may even equip the operating system with extra functions/APIs that gather binary, ternary, or quaternary projections and feed them to the safety-monitoring component.

When the period inference is used in runtime monitoring tools or time-debugger tools designed for a special system with known parameters, it is typically possible

to obtain richer projection types such as ternary and quaternary projections as we explained earlier. We will later (in Sect. 4) investigate how the extra information in these projections can be used to improve the accuracy of the period inference Problems 2 and 3.

4 Regression-based period mining

This section first introduces the challenges of the period inference (PI) problem and then presents our solution framework.

Challenges. As mentioned earlier, the PI problem has a long history in signal processing. Methods such as periodogram (Schuster 1898) and circular autocorrelation (or autocorrelation for short) (Gubner 2006) have been applied to infer periodicity of a signal and shown to work well in the presence of small (or standard) noise. However, they do not perform well (see Sect. 5) when applied to the PI problem because: (i) they may generate many period candidates most of which are irrelevant, (ii) although they assign a weight (called *power*) to each candidate, there is no direct relation between the weight and the true period, (iii) they cannot cope with preemptions well because they perceive each preemption as a new occurrence of the event under analysis (which adds a significant amount of noise to their inputs), and (iv) the true period is not necessarily among their generated candidates (specially in the autocorrelation method).

We then decided to look into the learning-based methods that could work well on the PI problem. We specially focused on those whose decision logic is *explainable* and traceable by a human. Therefore, we deliberately avoided using deep neural networks for the problem or for the feature extraction. However, this raised the next challenge: how to extract meaningful and helpful features from a projection? A starting point could be to use the whole binary projection as a feature and let the machine-learning method figure out the period. However, that could lead to two major issues: (i) dimensionality problems with the feature space, and (ii) having inputs with varying-length.

High-dimensional feature spaces typically lead to sparse data which in turn reduces the efficiency and increases the runtime of model learning (Bellman 1961). Moreover, most machine-learning methods require a fixed input size which implies that the input projection must be cut (unanimously for all projections of all training and testing task sets). However, since task sets have different hyperperiods, putting a predetermined cut-off threshold could either lead to low accuracy (if the cut-off is too short) or to a huge runtime and low efficiency in learning the model (if the cut-off is too long).

Solution highlights. The framework we propose to solve the period inference problem suggests a four-stage pipeline where Stage 0 extracts features and Stages 1 to 3 are for accuracy improvement of period estimation. Figure 2 shows the pipeline and the stages.

In Stage 0, we extract a fixed set of features from the top k highest-rank candidates of the periodogram and autocorrelation methods (see Sect. 3.1). In Stage 1, we use supervised-learning methods, and in particular, the regression-based machine

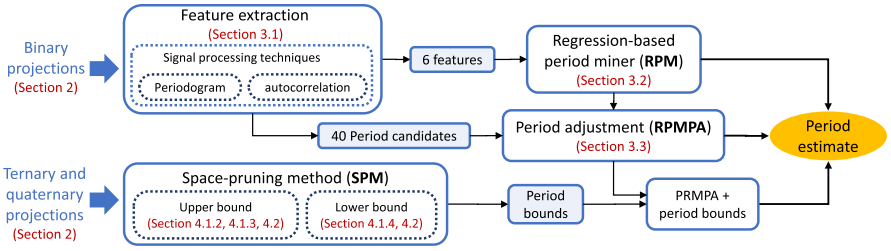

Fig. 2 Our period inference framework. The edges denote the information flow between our algorithms

learning (RBML) methods, to determine the relationship between our feature vectors and the target output, i.e., task period (see Sect. 3.2). RBML methods are commonly used when the goal is to predict a continuous output that takes order into consideration (in our case, the period). We call our RBML solution *regression-based period miner* (RPM). In Stage 2, we further adjust the predictions of RPM according to a set of high-ranked candidates from periodogram and autocorrelation. This aims to use RPM as a referee whose purpose is to highlight the most accurate peak from the two signal-processing methods (see Sect. 3.3). Finally, Stage 3 introduces some pruning rules using the extra information provided in ternary (Sect. 4.1) and quaternary (Sect. 4.2) projections to further restrict the number of candidates.

4.1 Feature extraction

Next, we explain our feature extraction and briefly introduce the periodogram and autocorrelation methods.

Periodogram (Schuster 1898). Consider the binary projection as a sequence P_i^B (where p_n is the nth item of the projection) and its discrete Fourier transform $X(f)$. The periodogram $\mathcal{P}$ gives an estimation of the spectral density of the discrete signal P_i^B and is obtained from the squared magnitude of the Fourier coefficients $X(f)$, as presented in Leondes (1996):

$$\mathcal{P}(f) = \frac{1}{N}\|X(f)\|^2, \tag{4}$$

where $N = |P_i^B|$ is the sequence length and $\mathcal{P}(f)$ is the power of frequency f. The Fourier coefficients $X(f)$ can be obtained from the sequence P_i^B as follows

$$X(f) = \sum_{m=1}^{N} p_m \cdot e^{-j\cdot 2\cdot \pi \cdot f \cdot m}. \tag{5}$$

The norm of a Fourier coefficient is the magnitude of that coefficient, namely, $\|X(f)\| = \sqrt{Re\{X(f)\}^2 + Im\{X(f)\}^2}$, where $Re\{X(f)\}$ and $Im\{X(f)\}$ are the real and imaginary coefficients for each frequency f, respectively.

Figure 3a and b show two periodograms obtained for two periodic tasks with period 1000 and 5000 from a task set with four tasks scheduled by rate monotonic

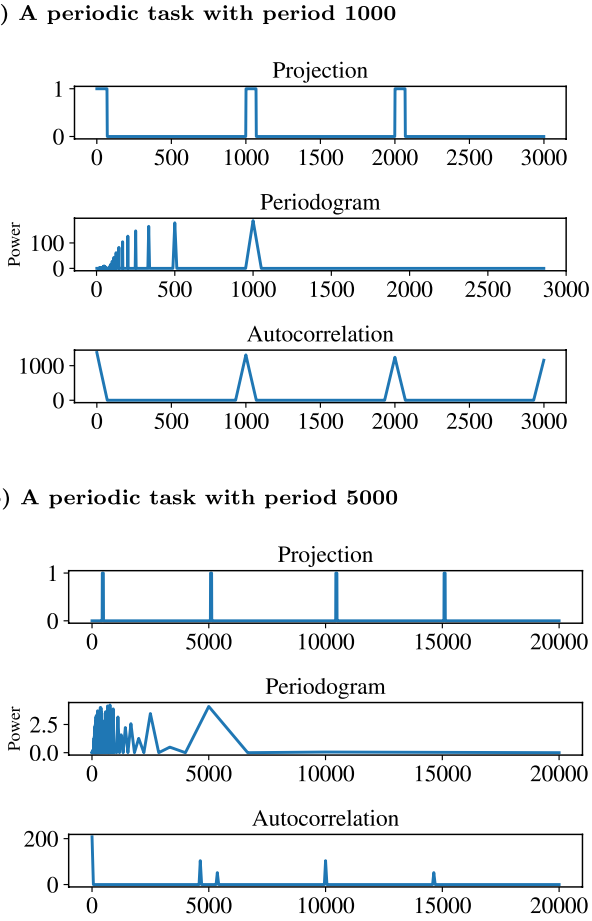

Fig. 3 Periodogram and circular autocorrelation methods applied on task projections for a task with **a** period 1000, and **b** period 5000 from a system containing 4 tasks, with a total utilization of 30% scheduled by rate monotonic. The other two tasks have a period of 2000 and 10000, respectively

scheduling policy. The horizontal axis shows the frequency values f and the vertical axis shows the power of each frequency, i.e., $\mathcal{P}(f)$.

As can be seen in Fig. 3a, the highest peak in this example (here, peak refers to a jump in the diagram) of the periodogram indicates the true period of the task, i.e., 1000. However, for the task with period 5000, this observation does not hold; the true period of this task is not the highest-peak but the 5th highest peak. The lower the priority of a task, the higher is the amount of interference it will have in its schedule. These interferences make the projections less regular and hence result in a more irregular periodogram that has many peaks.

Circular Autocorrelation (Gubner 2006). It is a metric that describes how similar is a sequence to its past values for different circular phase shifts. We use Vlachos et al. (2005) method to compute the circular autocorrelation:

$$\mathcal{ACF}(w) = \frac{1}{N}\sum_{n=0}^{N-1} p_w \cdot p_{n+w}, \tag{6}$$

where N is the sequence length and w is the phase shift. In the case of period inference problem, we would expect that the highest value of the autocorrelation function would be at a lag w equal to the true period.

A practical way to compute the $\mathcal{ACF}$ is to translate the operations into the frequency domain. Since (6) is a convolution, one can compute it with the dot product between the Fourier coefficients of the sequence and their complex conjugates (Vlachos et al. 2005):

$$\mathcal{ACF} = \mathcal{DFT}^{-1} X \cdot X^*, \tag{7}$$

In this paper, we apply the discrete Fourier transform on the projection and extract the Fourier coefficients using (5). Furthermore, we perform the dot product between the coefficients and their complex conjugates and apply the inverse Fourier transform on the result to obtain the autocorrelation. An implementation of our method can be found on github[2], along with the rest of the framework.

Figure 3a and b illustrate the usage of autocorrelation when the input is a projected trace. Firstly, we notice that the highest value that this technique exhibits is for a lag (period) of $w = 0$. This behavior is normal, since the highest similarity between a signal and itself is present when the two signals perfectly overlap with each other, e.g., at time 0 (see Eq. 6). Hence, the peak at 0 is excluded from the examination. The other observation is that, similar to the periodogram, the autocorrelation method is able to discover the true period only in the case from Fig. 3a, while for the second period, its top peak indicates an erroneous value. Moreover, we observe that the autocorrelation is sensitive to low utilization values.

In Fig. 3b, we see that the start time of the task with period 5000 is not at integer multiples of 5000 and varies a bit due to the interference caused by other high-priority tasks in the system. However, even though this task has not been preempted, we see that the projection does not have any overlap with itself when is shifted by the true period of 5000 (i.e., at $w = 5000$). As a result, the autocorrelation method could not detect the actual periodic behavior. However, it could observe two smaller peaks slightly shorter and slightly larger than 5000 at 4635 and 5365, respectively.

It is worth noting that, both periodogram and autocorrelation methods have an $O(N \log N)$ time complexity, where N is the length of the projection.

Extracting fixed-size features. Our fixed-size candidate list is constructed from the top $k = 3$ peaks of the outputs of the two methods, namely, we gather k-highest peaks from periodogram and k-highest peaks from the autocorrelation methods. It is worth noting that the width of a peak is correlated with the position of the peak in periodogram (the further from the origin the larger the width). Thus, it does not provide enough information to be considered as a feature for regression.

[2] https://github.com/SerbanVadineanu/period_inference.

Table 1 Overview of best performing families of regression algorithms and for each family the best model (Delgado et al. 2019)

Algorithm	Nickname	Category
Cubist Regression (Quinlan 1992, 1993, 2014)	*cubist*	Rule-based
Generalized Boosting Regression (Friedman 2002)	*gbm*	Boosting
Averaged Neural Network (Ripley 2007)	*avNNet*	Neural Networks
Extremely Randomized Regression Trees (Geurts et al. 2006)	*extraTrees*	Random Forests
Bayesian Additive Regression Tree (Chipman et al. 2010)	*bartMachine*	Bayesian Models
Support Vector Regression (Cortes and Vapnik 1995)	*svr*	Support Vector Machines

Having a feature set of size $k = 6$ allows us to work on a much smaller dimension for the input-data and have fixed input size to use with our regression-based solution. For the cases when there are fewer than k peaks for a method, the number of features is completed by appending the highest peak of that method until we reach the desired k. The choice on the number of features, i.e., $k = 3$, was made after evaluating the impact of k on various scenarios and finding out the suitable value that results in a high accuracy without increasing the dimensions of the feature space (Fig. 7b in Sect. 5 compares different choices).

4.2 Regression methods

Regression analysis is a method originating from statistics, whose purpose is to estimate the relationship between a dependent variable (or "outcome") denoted by Y and one or more independent variables (or "features") denoted by X. In machine learning, regression is employed when the aim is to predict a continuous output, which takes order into consideration. A regression model is formally described by

$$Y_i = f(X_i, \beta) + e_i, \tag{8}$$

where Y_i is the outcome variable, X_i is a feature vector, β represents unknown parameters, and e_i is an additive error term (residual) associated with the prediction.

Since we try to estimate the period from a projection, in our regression scenario, the dependent variable Y_i is the task's period T_i. The independent variables X_i contain the features we extracted at the previous step, while the function f comes from the choice of a regression algorithm, whose parameters β need to be estimated during the training phase.

In other words, our goal is to choose the form of function f and to compute the estimates of the parameters $\hat{\beta}$ such that the function has the best fit on the data. In order to assess how well the model fits the data, the predicted outcome, i.e., $\hat{Y}_i = f(X_i, \hat{\beta})$, is compared against the true dependent variable. The comparison is present in the shape of a *loss function* $L\left[Y, f\left(X, \hat{\beta}\right)\right]$, where Y is a vector containing the outcome variables and X includes all vectors of independent variables. For instance, the most commonly used loss function is the *mean square error* (MSE) (also used in our paper):

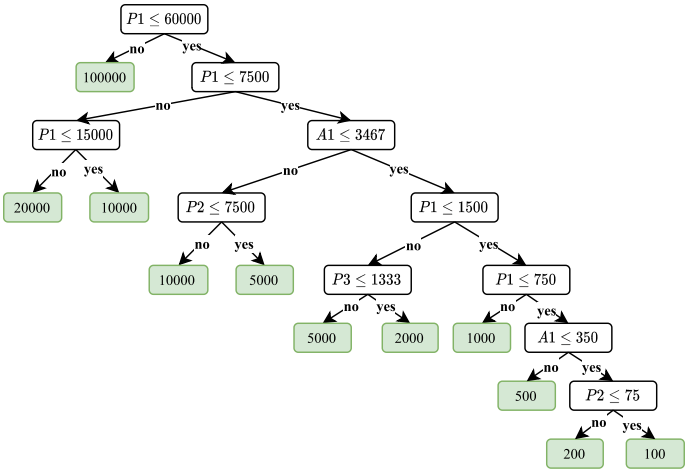

Fig. 4 A simple regression tree fitted on a data set with 4 tasks and a total utilization of 30% (see details in Sect. 5.1 for automotive tasks)

$$\text{MSE} = \frac{\sum_{j=1}^{N} (Y_j - \hat{Y}_j)^2}{N}, \tag{9}$$

where N is the total number of observations.

The choice of regression methods. Table 1 lists the overall best performing families of regression algorithms and for each family the best model, as suggested by Delgado et al. (2019) in their extensive recent survey on the performance and effectiveness of regression methods. These methods present distinctive characteristics in their implementation, namely, they do not theoretically dominate each other. Hence, in order to answer the question "which regression method performs best for the period-inference problem", we implemented and investigated all of these methods to gather insights about their performance on our particular problem.

We, however, anticipate to see that the tree-based solutions (*cubist*, *gbm*, *extraTrees*, *bartMachine*) have a better performance than *svm* and *avNNet* because we expect the transition from a set of candidate periods (the features) to the true period to be better approximated by a set of rules and/or comparisons rather than a linear or non-linear combination of these features as in *svr* and *avNNet*, respectively.

Regression trees. A majority of the RBML methods in Table 1 are *variations of* regression trees. A regression tree (Breiman et al. 1984) recursively partitions the feature space of the data into smaller regions until the final sub-divisions are similar enough to be summarized by a simple model in a leaf. This model can be simply the average of the outcomes from that sub-division.

Figure 4 shows the rules generated by a regression tree that was trained on the automotive task sets with four periodic tasks and 30% utilization (see details of the task set generation in Sect. 5.1). The features used for training are the three highest peaks from the periodogram (denoted by $P1$, $P2$, and $P3$) and autocorrelation (denoted by $A1$, $A2$, and $A3$) methods. The non-terminal nodes represent

the *rules* that will be used to guide the inference process by narrowing down the period estimate of a new task.

To make it more tangible, we explain how to use the regression tree in Fig. 4 to estimate the period of the two tasks in Fig. 3a and b. In the first step, we derive the three highest peaks of the periodogram and autocorrelation methods to build the feature vectors X_1 and X_2 for the first and second tasks, respectively. Here, $X_1 = \langle P1{=}1000, P2{=}500, P3{=}333, A1{=}1000, A2{=}2000, A3{=}3000\rangle$ and $X_2 = \langle P1{=}769, P2{=}666, P3{=}5000, A1{=}4635, A2{=}10000, A3{=}5365\rangle$. Next, we traverse the tree by evaluating the rules starting from the root node. For example, for the first task, $P1 = 1000$ and hence the condition in the root node (i.e., $P1 \leq 60000$) is satisfied. Thus, we go to the right branch and repeat the process until we reach to a leaf. The value in the leaf is the period estimate. In this example, the trained model can accurately estimate both tasks' period.

An interesting observation in Fig. 4 is the exclusion of $A2$ and $A3$ in the tree's rules which basically means that these two features had no impact on the final period estimate. With a further investigation, we observed that typically in task sets with low utilization, the trained regression trees tend to be smaller and rules contain fewer features because there are less preemptions (and hence, less noise) in the input. However, with an increase in utilization, the tree is forced to consider more features and even become deeper to keep the estimation error low.

Training a regression tree can be done in $O(m \cdot N \cdot \log N)$, where m is the number of features (in our case it is a constant value equal to 6) and N is the number of samples (projections) used for training. Later in Sect. 5, we provide an evaluation on the runtime and memory consumption of various RBML methods.

Understanding how a simple regression tree works, we can now discuss the actual RBML methods used in our work according to the suggestions of Delgado et al. (2019). Note that four of these methods are extended variations of the regression trees but none is as simple as the tree shown in Fig. 4.

Cubist Regression (cubist). Kuhn and Quinlan 2020; Quinlan 1992, 1993, 2014). It is a regression tree whose leaves embed linear regression models instead of simple 'estimates of the output'. The tree can be further reduced by combining or pruning the rules via collapsing the nodes of the trees into rules.

By training a *cubist* regression model on the same data-set as in Fig. 4, we obtain the following rules:

1. **If** $(A1 \leq 2000)$ **then** return $P1$,
2. **If** $(P1 \leq 1250 \wedge A1 > 2000)$ **then** return 5000,
3. **If** $(P1 > 1250)$ **then** return $P1$.

In this example, we observe that while the rules and outputs rely on the top candidates of the periodogram, they are not limited to them. For example, rule 2 outputs the period 5000 which is not among the three top features of periodogram. The *cubist* regression uses these rules to compensate for projections where the periodogram is wrong.

Cubist regression consumes notably less memory than the regression trees (see Sect. 5) and hence it is a better choice when the solution must have low memory consumption and runtime. However, we also noticed a growth in the number of rules when it is trained on task sets with high utilization because then the underlying regression tree from which the cubist regression rules are obtained gets larger and deeper when the number of preemptions increases.

Generalized Boosting Regression (gbm) (Greenwell et al. 2019; Friedman 2002). This algorithm is a regression tree-based solution which uses a committee of regression trees of fixed size. The initial prediction of the algorithm starts from a leaf, which contains the average value of the outcome variables (i.e., the periods). The next step is to compute the residuals of this initial prediction against the true output (true period). Next, a regression tree is fitted on the data, but having the previously computed residuals as the outcome variables.

In order to preserve the generalization capabilities of the model, the results from the tree are multiplied by a constant value. Afterwards, the output from the tree is added to the initial leaf to obtain a new set of predictions, which are again used to compute residuals. The process is repeated until a maximum number of trees is reached.

Extremely Randomized Regression Trees (extraTrees) (Simm et al. 2014; Geurts et al. 2006). The algorithm relies on a committee of regression trees for its predictions. When building the trees, this method randomly picks a rule for each feature (instead of searching for a rule that minimizes the error) and then chooses the one that provides the lowest error. Hence, a randomized regression tree is much faster to build than a regular regression tree.

Bayesian Additive Regression Tree (bartMachine) (Kapelner and Bleich 2016; Chipman et al. 2010). Similar to *gbm*, this method also relies on a group of trees, where each tree is fit on the residuals of the predictions from a previous tree. The major difference is that *bartMachine* is based on a probability model containing a set of priors for the tree structure and a likelihood for the leaves' values. *extraTrees*, *gbm*, and *bartMachine* stop building the model when a given (maximum) number of trees is achieved.

Averaged Neural Network (avNNet) (Kuhn 2020; Ripley 2007). The technique involves a committee of five multilayer perceptrons having the same size, but trained using different random seeds. The network is set to have linear output neurons, which makes it suitable for regression. Finally, the predictions from the five networks are averaged to provide the final estimate.

Support Vector Regression (svm) (Meyer et al. 2019; Cortes and Vapnik 1995). The goal of *svr* is to find a line or a hyperplane that is able to fit the most data points within a certain margin from it. Moreover, it can accommodate non-linear trends by fitting the line in a transformed feature space using a kernel function.

Sections 5 and 6 provide further insights on the performance of the RBML methods.

4.3 Candidate selection

As the example in Fig. 3 shows, the true period is among one of the peaks of the periodogram and autocorrelation, although not always is the highest peak. After further investigations, we observed that on the one hand, in a majority of projections, the true period is indeed among the peaks of periodogram and autocorrelation. However, it is hard to know which of those peaks just by looking at their power or rank. On the other hand, the RBML methods typically predict only an approximation of the real period which is not always equal to the true one (resulting in non-zero errors in most cases). Thus, we introduce a further pruning phase on the output of our RPM method and create a method called *RPM with period adjustment* (RPMPA).

RPMPA treats the RPM method as a referee which chooses the right period from a set of candidates. Namely, it first calculates the period estimate using the RPM method and then finds the closest period to this estimate from a fixed set of values gathered from the 20 highest peaks of each of the periodogram and autocorrelation (hence, 40 candidates in total). The number of candidates (i.e., 40) is a hand-tuned value and comes from experimenting on many task sets (see Sect. 5.2).

5 Deriving period bounds to improve accuracy

Why. As it will be shown in our experiments, despite the success of the RPMPA method to improve accuracy, in some scenarios, the "adjustment" step increases the error instead of reducing it (see Sect. 5). Those cases happen when the underlying regression algorithm (as a part of the RPM method) produces an output that significantly deviates from the true period. As a result, when the RPMPA chooses a candidate, it introduces more error. To reduce the chance of deviating from the true period, this section presents methods to derive upper and lower bounds on the period directly from the input projections so that the search space for RPMPA is further narrowed down and its final error is reduced.

What. We present a *space-pruning method* (SPM) whose goal is to derive a lower and an upper bound on the possible set of period values by looking at the higher-order projections such as ternary and quaternary projections. These bounds are meant to remove the impossible period values from the candidate set generated from the highest 20 peaks of each of the periodogram and autocorrelation methods before they are fed to the RPMPA (recall Fig. 2).

It is worth noting that if applying the lower and upper bounds on the 40 period candidates results in an empty set (i.e., all 40 candidates are outside of the bounds), we suggest to just use the upper bound as the period estimate. Later in Sect. 5.3 (Fig. 9), we show that choosing the upper bound results in higher accuracy than just using the output of the RPM (regression) method.

How. Ternary and quaternary projections include information about the *idle times* and the execution of *lower-priority tasks*, respectively. These information together with some basic knowledge about the scheduling policy can help deriving upper and lower bounds on the actual periods. For example, under a work-conserving scheduling policy, we can deduce that "*if a task has accessed the resource between two idle*

times in a ternary projection, then it must have released a job somewhere between those idle times". In the example shown in Fig. 1d, at least one job of τ_3 must have been released in the interval [10, 19) since there is at least a '1' in the ternary projection of τ_3 during this interval. Similarly, another job must have been released in the interval [20, 25). An upper bound on the period of this task can be derived from the largest inter-arrival times observed in the projection. In Sects. 4.1.2 and 4.1.3, we will elaborate on how to derive such upper bound when tasks do not have or have release jitter, respectively.

5.1 Improving the accuracy for ternary projections

Our key idea to derive an upper bound on the task's period is to traverse the ternary projection to find pairs of consecutive intervals separated by idle times in which the task has occupied the resource. We call them *effective intervals*. Then by looking at every three consecutive effective intervals, we can obtain *one upper bound* on the task's period. After traversing the whole projection, *the smallest upper bound* found is the bound we use to prune the period candidates obtained from the peaks of periodogram and autocorrelation.

In the rest of this section, we first discuss how to obtain the effective intervals (see Sect. 4.1.1), and then how to derive upper bounds for tasks with no release jitter (see Sect. 4.1.2) and with bounded release jitter (see Sect. 4.1.3). It is worth noting that our upper bounds for the period are tighter than that of Vădineanu and Nasri (2020). Finally, in Sect. 4.1.4, we show how to calculate a lower bound on the period.

5.1.1 Extracting effective intervals from ternary projections

Assumptions (to derive the upper bounds). Before we explain how to obtain the upper bounds, we summarize the required assumptions: **(A1)** the scheduling policy is work-conserving and **(A2)** the task under analysis does neither skips a job (the BCET of the task is not zero) nor suspends itself. If these assumptions do not hold, then the upper bound is ∞. In practice, it is easy to check if the scheduling policy that governs the resource is work-conserving. Most well-known scheduling policies implemented by operating systems are work conserving, for example, EDF, fixed-priority scheduling, FIFO scheduling, etc. To check if the assumption A2 holds, one may use a separate monitoring tool that checks whether each instance of the task has been completed. If the code of the task is available, an easier solution is to instrument the task so that it sends a signal whenever it finishes. If no 'missed' job occurs during the time the projection is being stored, then the upper bounds that we derive in Sects. 4.1.2 and 4.1.3 can be used.

Let P_i^T be a ternary projection and x be a time instant at which $p_x = 1$ and $\exists z < x$ in the ternary projection such that $p_z = idle$. Then, the beginning of the effective interval that contains the time instant x, called the *effective point* (denoted by $I^s(x, z)$), is a function that returns the latest idle-time prior to the execution of τ_i, namely,

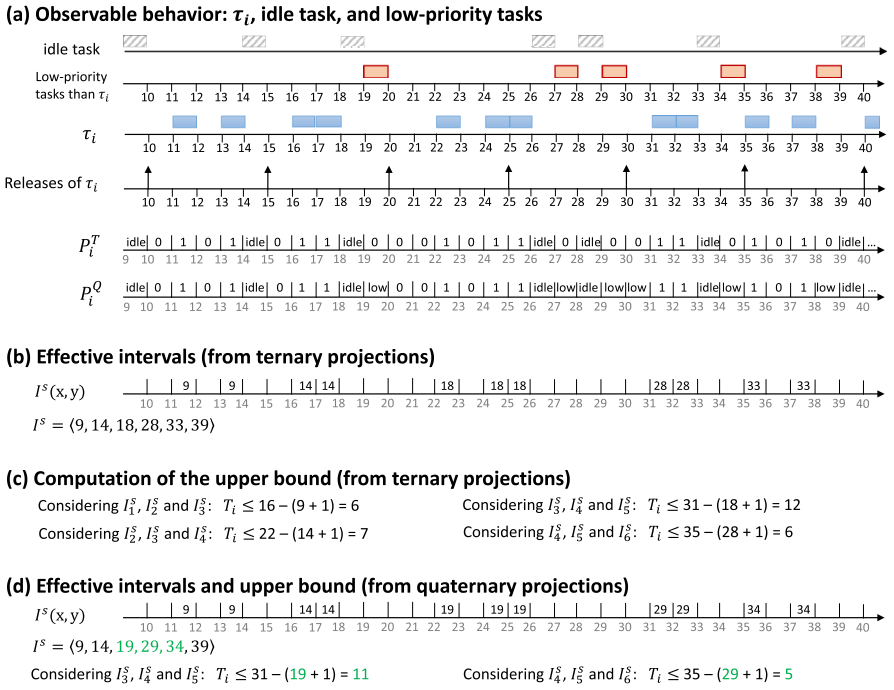

Fig. 5 Observation from the execution of a periodic task τ_i with no release jitter, $T_i = 5$, and $C_i^{min} = 1$ and $C_i^{max} = 2$. **a** Ternary and quaternary projections, **b**, **c** effective intervals and upper bound calculation from ternary projections, **d** effective intervals and upper bound calculation from quaternary projections

$$I^s(x, z) = \max\{k |\ z \leq k < x \wedge p_k = idle \wedge \forall p_y, k < y < x, p_y \neq idle\}. \quad (10)$$

Note that the effective points are only defined for time slots x in which $p_x = 1$. By traversing through the projection once, one can obtain the starting points of all effective intervals.

In the example shown in Fig. 5a and b, the effective points are $I^s = \langle 9, 14, 18, 28, 33, 39 \rangle$. Note that when calculating $I^s(31, 18)$, the idle slot at time 26 does not have the conditions of Eq. (10) because there exists another idle slot in a later time than 26, i.e., at time 28. If it is not certain that the starting point of the ternary projection was aligned with an idea time, the first idle slot in the projection will be considered as the first effective point.

Effective intervals are obtained by considering consecutive pairs of items in I^s, namely, $[I_1^s, I_2^s)$, $[I_2^s, I_3^s)$, $[I_4^s, I_5^s)$, etc.

5.1.2 Deriving an upper bound for tasks with no release jitter

We start with a case where the task under analysis does not have release jitter. Later (in Sect. 4.1.3), we will extend our discussions to tasks with bounded release jitter.

Let $I^s_{j-1}, I^s_j, I^s_{j+1} \in I^s$ be three consecutive effective points in the ternary projection P^T_i. In order to obtain an upper bound on the period, we calculate the largest possible distance between the release of two consecutive jobs of the task that have been released in the intervals $[I^s_{j-1}, I^s_j)$ and $[I^s_j, I^s_{j+1})$. To achieve this goal, we will calculate the earliest possible release (denoted by $e(I^s_{j-1}, I^s_j)$) of a job of τ_i that is released in the interval $[I^s_{j-1}, I^s_j)$ and the latest release time of *the first job of* τ_i *that is released in the interval* $[I^s_j, I^s_{j+1})$ (denoted by $l(I^s_j, I^s_{j+1})$).

Since the ternary projections do not include any special information that allows us to distinguish two jobs of the same task from each other, and since we have no knowledge about the execution time of the task, apart from that the BCET is not zero (i.e., the task does not skip a job), the earliest time at which a job of τ_i might have been released in the interval $[I^s_{j-1}, I^s_j)$ is:

$$e(I^s_{j-1}, I^s_j) = I^s_{j-1} + 1. \tag{11}$$

To obtain the $l(I^s_j, I^s_{j+1})$, we find the earliest time at which a job of task τ_i has occupied the resource in the interval $[I^s_j, I^s_{j+1})$. Namely,

$$l(I^s_j, I^s_{j+1}) = \min\{k \mid I^s_j < k < I^s_{j+1} \ \wedge \ p_k = 1\}. \tag{12}$$

Note that there might be more than two jobs that have been released in the interval from $[I^s_{j-1}, I^s_j)$. For example, in Fig. 5a and b, the interval [18, 28) contains two actual jobs of τ_i (released at time instants 20 and 25) but we assume there is only one (which is released at time $e(18, 28) = 19$) since we have no evidence in the projection that suggests that the occupation of the resource at the time slot 25 belongs to a new job of τ_i.

The next step is to obtain an upper bound on T_i using the difference between $e(I^s_{j-1}, I^s_j)$ and $l(I^s_j, I^s_{j+1})$:

$$T_i \leq l(I^s_j, I^s_{j+1}) - e(I^s_{j-1}, I^s_j). \tag{13}$$

The following theorem proves that Eq. (13) is a sound upper bound for the period.

Theorem 1 *Given two consecutive effective intervals* $I_{j-1} = [I^s_{j-1}, I^s_j)$ *and* $I_j = [I^s_j, I^s_{j+1})$ *for a task* τ_i *that does not have release jitter, Eq.* (13) *provides a safe upper bound on the period of the task.*

Proof The proof is trivial. By the definition of effective points, we know that there is at least one time instant in each of the intervals I_{j-1} and I_j at which task τ_i has occupied the resource. Since the scheduling policy is work conserving and at time I^s_{j-1} the resource was idle, the earliest time at which a job of task τ_i could have been released in the interval I_{j-1} is at $I^s_{j-1} + 1$ (calculated by Eq. 11). Moreover, from

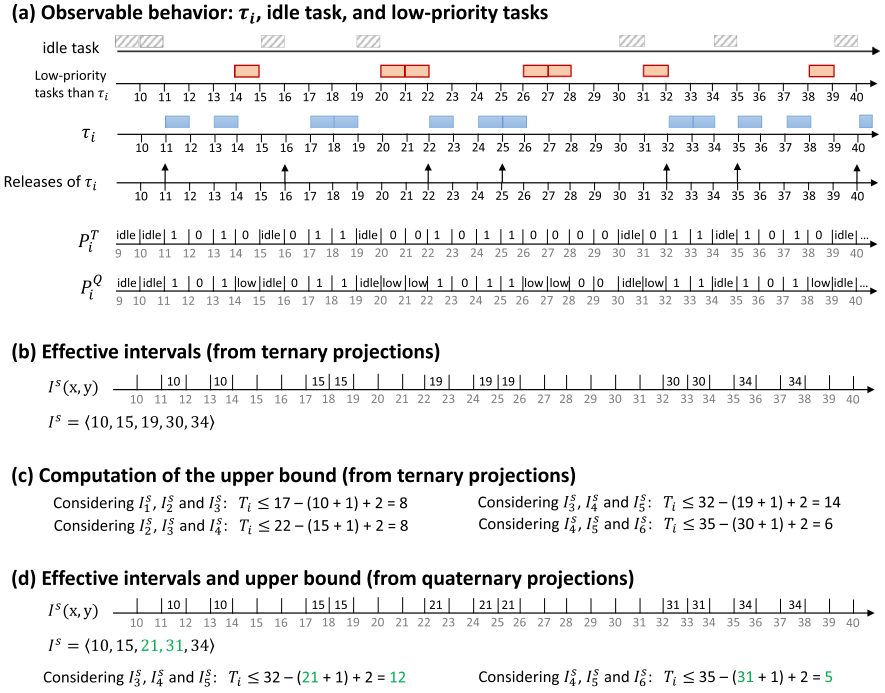

Fig. 6 Observation from the execution of a periodic task τ_i with release jitter, $T_i = 5$, $\sigma_i = 2$, and $C_i^{min} = 1$ and $C_i^{max} = 2$. **a** Ternary and quaternary projections, **b**, **c** effective intervals and upper bound calculation from ternary projections, **d** effective intervals and upper bound calculation from quaternary projections

Eq. (12), we know that $l(I_j^s, I_{j+1}^s)$ is the earliest instant at which the task has occupied the resource within the interval I_j. Hence, the latest release of the first job of the task within this interval must have been at or before $l(I_j^s, I_{j+1}^s)$. Consequently, the distance between two releases of the task τ_i in the intervals I_{j-1} and I_j cannot be larger than $l(I_j^s, I_{j+1}^s) - e(I_{j-1}^s, I_j^s)$. Hence, Eq. (13) provides a safe upper bound on the period of τ_i. □

Figure 5c shows how to calculate four upper bounds for T_i from different sets of effective intervals in the example shown in Fig. 5a. These upper bounds are 6, 7, 12, and 6. Thus, $T_i \leq 6$ is the tightest upper bound that our SPM method obtains from the ternary projections for this example.

It is worth noting that the difference between the upper bound in the current paper and in our prior work (Vădineanu and Nasri 2020) is that here, we calculate the latest arrival time of the *first* job of the task in the interval $[I_j, I_{j+1})$ but in our prior work, we calculated the *last* arrival time of a job of the task in the interval $[I_j, I_{j+1})$. This has been captured by Eq. (10) in Vădineanu and Nasri (2020) as follows $fin(I_j, I_{j+1}) = \max\{k \mid I_j^s < k < I_{j+1}^s \ \wedge \ p_k = 1 \ \wedge \ \forall p_y, k < y < I_{j+1}^s, p_y \neq 1\}$.

As it can be seen, $fin(I_j, I_{j+1})$ produces a value that is always larger than or equal to $l(I_j, I_{j+1})$ (defined in Eq. 12). Hence, the new upper bound in Theorem 1 is always smaller than or equal to the upper bound in Vădineanu and Nasri (2020).

5.1.3 Deriving an upper bound for tasks with bounded release jitter

When a task has release jitter, Eq. (13) may not hold anymore. For example, by applying Eq. (13) on the intervals [30, 34) and [34, 39) in the example shown in Fig. 6a (which represents a periodic task with at most two units of positive release jitter, i.e., $\sigma_i = 2$), one may mistakenly conclude that the period must be smaller than or equal to 4 because $e(30, 34) = 31$ and $l(34, 39) = 35$. However, by looking at the actual release times of the task, we see that the idle slot at time 30 is caused by the release jitter of a job of τ_i that has been released at time 32 instead of 30.

Assumptions and requirements. To be able to derive an upper bound on the period of a task that has release jitter, we would need to know the maximum amount of the release jitter that the task may suffer (i.e., σ_i). Such information is typically available when the period inference framework is used for runtime monitoring of a known system. If the exact value of σ_i is not know, it is fine to use a safe upper bound on it, if available.

If no safe upper bound on the maximum release jitter can be provided, then only RPM and RPMPA (but not the SPM) solutions can be used to estimate the period. As we will see later in Sect. 5.3, these two solutions can accurately predict the period even when there is release jitter.

Solution idea. Deriving an upper bound for T_i requires finding an upper bound on the largest distance between arrival times of two consecutive jobs of the task, where the arrival time is the *expected release time* when there is no release jitter[3]. We will derive the latter upper bound by calculating a lower bound on the arrival time of a job released in the effective interval I_{j-1} and an upper bound on the arrival time of the next job released in the effective interval I_j.

From the definition of the effective intervals, we know that the task has occupied the resource during the interval $[I^s_{j-i}, I^s_j)$. The earliest time at which a job of the task could actually be released in this interval is at $I^s_{j-1} + 1$ (since the resource was idle at I^s_{j-1}). However, in the presence of release jitter, the arrival time of that job could be earlier than $I^s_{j-1} + 1$. By reducing the maximum value of release jitter, i.e., σ_i, from the release time, we will have a safe lower bound on the *earliest possible arrival time* of that job at $I^s_{j-1} + 1 - \sigma_i$.

Since we consider positive release jitter, the actual release time of a job is already an upper bound on the arrival time of that job. Hence, we can use Eq. (12) to obtain an upper bound on the arrival time of the "next" job of the task (in the effective interval $[I_j, I_{j+1})$). Hence, the new upper bound on the period of a task with at most σ_i units of release jitter is

[3] We follow Audsley's definition for arrival time. Please have a look at Sect. 2.1 for details.

$$T_i \leq l(I_j^s, I_{j+1}^s) - e(I_{j-1}^s, I_j^s) + \sigma_i. \tag{14}$$

Theorem 2 *Given two consecutive effective intervals* $I_{j-1} = [I_{j-1}^s, I_j^s)$ *and* $I_j = [I_j^s, I_{j+1}^s)$ *for a task* τ_i *that has at most* σ_i *units of positive release jitter, Eq.* (14) *provides a safe upper bound on the period of the task.*

Proof The proof is trivial and follows from the above discussion. □

Fig. 6c shows the calculations of the upper bound for the example shown in Fig. 6a. As it can be seen, there are four upper bounds for T_i and the tightest one is 6.

5.1.4 Calculating a lower bound for period

Assumptions. To obtain a lower bound, we would need the following assumptions: **(A1)** the scheduling policy is work-conserving, **(A2)** the task under analysis does not skip a job (e.g., there is no execution path in the task that has zero execution time and the activation of this task is not conditional to some external events) and the task does not self-suspend, **(A3)** the task has a constrained deadline and the projection does not contain any deadline misses.

If any of these assumptions do not hold, the lower bound on the period will be 0. Note that A3 can be known, for example, in systems that are equipped with separate monitoring tools that report any deadline miss or dropped jobs to the period-inference tool.

Key idea. To obtain a lower bound on period, we extract the largest interval with length L in which the task τ_i does not occupy the resource. This can be obtained as follows

$$\begin{aligned} L = \max\{b - a - 1 \mid 0 \leq a < b \leq |P_i^T| \ \wedge \\ p_a = p_b = 1 \ \wedge \ \forall j, a < j < b, p_j \neq 1\}. \end{aligned} \tag{15}$$

Equation (15) finds the largest interval $[a, b]$ in the projection between two time instants a and b, such that the task under analysis has occupied the resource at time a and b but not between them, i.e., $p_a = p_b = 1$ and $\forall j, a < j < b, p_j \neq 1$.

Under assumptions A1, A2, and A3, the length of the largest interval during which no job of the task τ_i has occupied the resource is upper bounded by $|L| \leq 2 \cdot T_i$. The reason is that in the worst case, the *largest interval during which the task does not occupy the resource* happens when one job of the task is executed right after its *arrival time* and the next job completes right before its deadline, resulting in a value slightly smaller than $2T_i$. Given that periodogram can contain many peaks at small periods, having a lower bound can help reducing the error efficiently. Note that this bound holds whether the task has release jitter or not:

$$T_i > 0.5\, L. \tag{16}$$

Theorem 3 *Given an interval $g = (a, b)$ during which τ_i is not present on the projection, i.e., $p_{a-1} = p_b = 1 \wedge \forall j,\ a \leq j < b,\ p_j \neq 1$, if assumptions A1, A2, and A3 in Sect. 4.1.4 hold, then $T_i > 0.5\ |g|$ is a safe lower bound on the period of the task.*

Proof The proof is trivial and follows the above discussion. According to A2 and A3, the task under analysis does not miss a job and does not have a tardy job, hence, its earliest finish time is when it starts its execution right at its arrival time and it has at most one unit of execution. From A3, we know that the latest theoretical upper bound on the completion time of a job is when it completes at its deadline. Since A3 assumes a constrained deadline, an upper bound on the deadline is the period of the task.

Now putting these two facts together, the largest interval during which a job of the task does not appear on the projection happens when one job completes as early as possible, i.e., if it is supposed to arrive at t_1, it completes at $t_1 + 1$ and the 'next' job completes as late as possible, i.e., at $t_2 = (t_1 + T_i) + T_i$. Consequently, the largest interval during which the task is not executing is upper bounded by $t_2 - t_1 = 2T_i$, in the worst case. Hence, if an interval g is found during which the task does not occupy the resource, $|g| < 2T_i$ because otherwise the task must have had a deadline miss (which would violate A3). □

It is worth noting that any interval $g = (a, b)$ that is consistent with Theorem 3 can be used to derive a lower bound on T_i regardless of what has occupied the resource during that interval (i.e., the resource might be idle or executing some tasks other than τ_i). However, such a lower bound might be too small (and hence ineffective). For example, one lower bound that can be obtained from Fig. 6a is for the interval [36, 37) which will result in $T_i > 0.5$. Obviously, it is less effective than the lower bound that is obtained from interval [26, 32) which results in $T_i > 2.5$.

Since both the lower bound and the upper bound can be calculated at the same time (by passing through the projection only once), they have a linear time complexity w.r.t. the projection length.

5.2 Improving the accuracy for quaternary projections

Key idea. Quaternary projections contain information about the intervals during which the lower-priority tasks were occupying the resource. Under a fixed-priority scheduling policy, we know that if a lower-priority task is executing, then the task under the analysis must have been completed (otherwise, the assumption about the scheduling policy will be violated). As a result, we can treat the moments/intervals during which a lower-priority task has occupied the resource as "idle instants" when calculating the upper bound on the period.

More formally, when obtaining the upper bound on the period of a task, it is possible to create an augmented ternary projection $P_i^{T'}$ from a quaternary projection P_i^{Q} using filter f that converts the '*low*' symbols in the quaternary projection to '*idle*' symbols in the augmented ternary projection, defined as follows:

$$f(p_j) = \begin{cases} 1, & p_j = 1 \\ idle, & p_j = idle \vee p_j = low \\ 0, & \text{otherwise} \end{cases}. \tag{17}$$

Definition 4 An *augmented ternary projection* $P_i^{T'}$ derived from a quaternary projection P_i^Q for task τ_i is defined as $P_i^{T'} = \langle f(p_j) \mid \forall j,\ 1 \leq j \leq |P_i^Q| \rangle$, where $f(p_j)$ is obtained from Eq. (17).

To use quaternary projections to derive an upper bound on the period, we need the following assumptions: **(A1)** the scheduling policy is work-conserving, **(A2)** the task under analysis does not skip a job (the BCET of the task is not zero) or self-suspend, and **(A3)** the system is scheduled by a preemptive fixed-priority scheduling policy.

Lemma 1 *Under the assumptions A*1, *A*2, *and A*3 (*Sect*. 4.2), *at any time slot at which the processor is idle or a task with a lower priority than* τ_i *has occupied the resource*, *the task* τ_i *cannot have a pending job*.

Proof The proof is a direct conclusion of scheduling the task set with a work-conserving preemptive fixed-priority scheduling policy and the fact that the task under analysis does not suspend itself and does not skip a job. Namely, whenever it is released, no other low-priority task can occupy the resource. Hence, if a low-priority task has occupied the resource, the task under analysis must not have a job in the ready queue (a job that has been released but has not completed). □

Lemma 1 allows us to treat augmented ternary projections (Definition 4) as a normal ternary projection when deriving the effective intervals.

Figures 5d and 6d show the effective points obtained from the augmented ternary projections and their impact on tightening the upper bound on the period. Later in Sect. 5.7 we will empirically compare the bounds obtained from ternary and quaternary projections.

6 Empirical results

We performed a set of experiments to answer the following questions: (i) Does our framework improve the accuracy w.r.t. the state of the art? (ii) How do various families of RBML methods compare against each other? (iii) How robust is our solution against uncertainties and non-deterministic events? (iv) What are the tradeoffs between the accuracy, runtime, and the memory requirements of various RBML methods? and (v) How good our solution generalizes to systems that are widely different from those on which it trained? Questions (i) and (ii) are addressed throughout

the evaluation section. Question (iii) is answered in Sect. 5.4, and finally, Sect. 5.6 focuses on questions (iv) and (v).

We divided our task systems into three groups: periodic task systems where every task is periodic but tasks might have release jitter or execution time variation (Sects. 5.3 and 5.6), non-periodic task systems, where the task under analysis is periodic but the rest of the system might not be periodic (Sect. 5.4), and case studies from actual systems (Sect. 5.5). The source code and our evaluation framework for these experiments are both available on *github* (Vădineanu 2020).

6.1 Experimental setup

For the experiments in Sects. 5.3, 5.4, and 5.6, we considered two types of task sets: automotive benchmark application and synthetic task sets. For the automotive benchmark applications, we adopted the model proposed by Kramer et al. (2015) for task sets used in automotive industry, where task periods are chosen randomly from {1, 2, 5, 10, 20, 50, 100, 200, 1000}ms with a non-uniform distribution provided by Kramer et al. (2015). For simplicity, we refer to the traces of these task sets as *automotive* traces.

Our synthetic task sets are comprised of non-harmonic periods. In order to ensure that the chosen periods cover evenly all magnitudes, we used a log-uniform distribution as suggested and described by Emberson et al. (2010). The periods are thereby generated for the range [100, 10000] with a base period of 100ms. For simplicity, we refer to the traces of these task sets as *log-uniform* traces. We use Stafford's Randfixedsum algorithm which is also used by Emberson et al. (2010) to generate random utilization values for the tasks and then use the utilization and the period to calculate the WCET of each task.

To generate the traces, we use *Simso* (Chéramy et al. 2014), an open source and flexible simulation tool that generates schedules under various scheduling policies and setups.

Evaluation strategy. The data set used for training the regression models is composed of the projections from 2000 traces (we saw no benefit in increasing the data set size in our preliminary experiment). The length of a trace is set to be either six hyperperiods (traces without random variations) and ten hyperperiods (when there is execution time variation or release jitter) for the experiments in Sects. 5.3 to 5.5. The same trace lengths are used for the testing to capture enough random behavior. In Sect. 5.6, we specifically investigate the impact of trace length on the accuracy of testing.

Metric. The metric we use to evaluate the accuracy is the average error, which is the mean of the individual errors a method makes for every period in a test set unless it is explicitly stated that the error has been obtained for only one task in the task set. Furthermore, we calculate the error of one experiment (that includes 2000 task sets) by using *fivefold cross-validation*. Namely, we divide the data set into five randomly chosen subsets of equal size. Out of the five subsets, four are used for training and one is used for testing. We measure the error of the testing and repeat the process until all five subsets have been used once for testing.

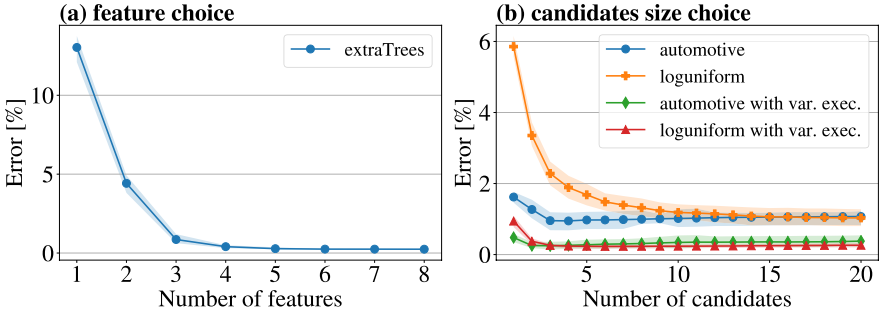

Fig. 7 The impact of the number of features (for RPM) and the number of candidates (for RPMPA) on the solution's accuracy. Note that the shade around the curves represents the confidence intervals for 0.95 confidence level

Baselines. We considered three baselines: (i) PeTaMi, a mining algorithm for periodic tasks (Iegorov et al. 2017), (ii) periodogram (Schuster 1898), and (iii) autocorrelation (Gubner 2006). PeTaMi represents the state of the art on period inference in the real-time systems community, while the other two represent widely used solutions from the signal-processing literature. These two were chosen to evaluate the improvements made by our RPM and RPMPA over solutions that are (only) based on signal-processing techniques.

We compare the RBML methods mentioned in Table 1, denoted by *cubist* (Quinlan 2014), *gbm* (Friedman 2002), *avNNet* (Ripley 2007), *extraTrees* (Geurts et al. 2006), *bartMachine* (Chipman et al. 2010), and *svr* (Cortes and Vapnik 1995). Each of these methods is defined by a set of hyperparameters that require tuning for improving the model's fit on the data. Hence, we performed an additional tuning phase using random search on the parameter's space. This step was integrated in the cross-validation process such that every training set comprised of the four subsets, is further split into a training and validation set. The parameters are varied while being trained on the training set and the model's performance is estimated on the validation set. The purpose of doing one more split is to avoid bias by not involving the test set into the parameter choice.

To be able to focus on the accuracy of the RBML methods, we only show the results of RPM method in Figs. 7, 8a to o, 12, and 15. We compare the accuracy of RPM with RPMPA in Figs. 8p to r and 9; Tables 2 and 4. We performed our evaluation on a Dutch supercomputer based in the cloud. We used thin nodes with 2×16 -core 2.6GHz Intel Xeon E5-2697A v4 (Broadwell) and 64GB of memory.

6.2 Parameter tuning

Before evaluating our solutions, we need to determine their parameters, i.e., the number of features for RPM and the number of candidates for RPMPA, since they impact the solution's accuracy. The evaluation from Fig. 7a was performed on an aggregated data set, containing automotive traces with four levels of utilization (0.3, 0.5, 0.7, and 0.9). Similarly, for the second experiment from Fig. 7b, we used data

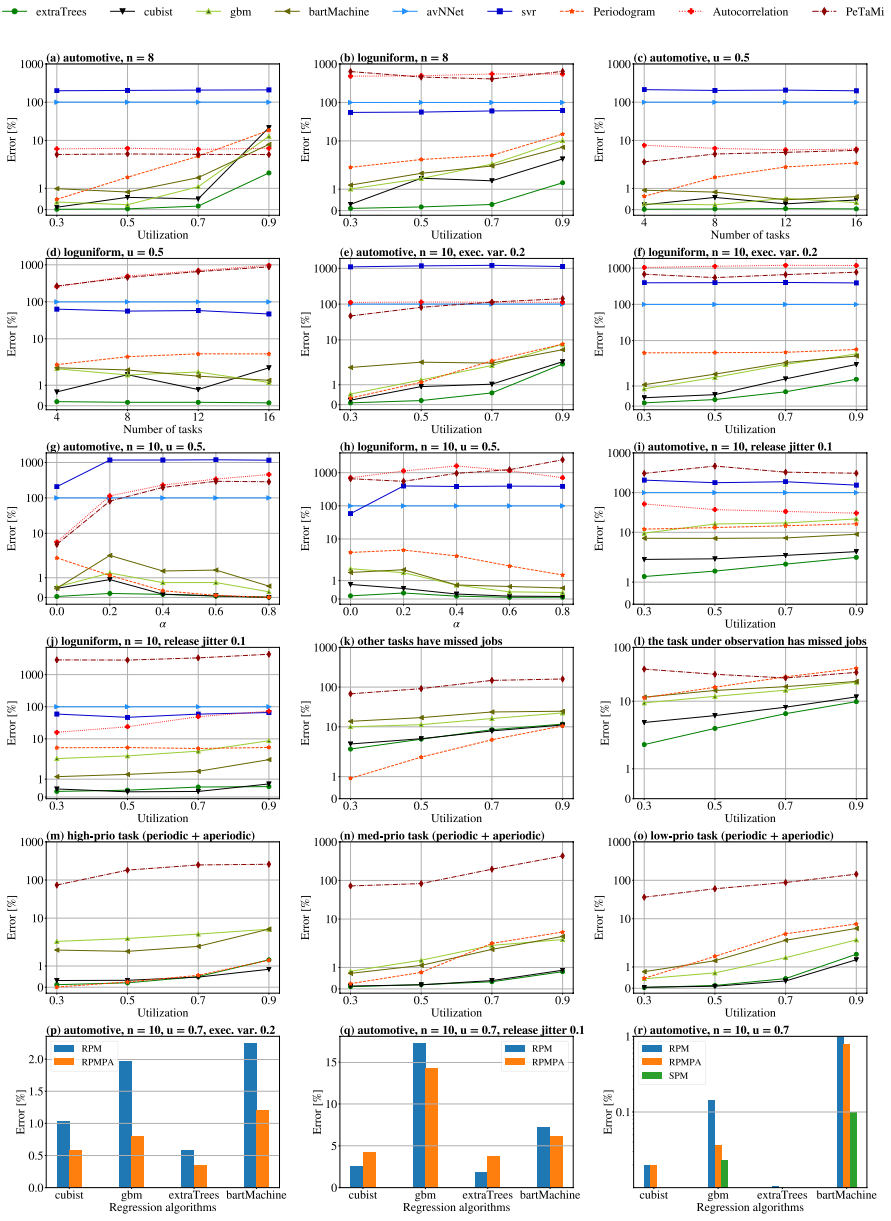

Fig. 8 Experimental results for periodic and preemptive systems

sets incorporating the four utilization values and we also kept 20% execution time variation for the tasks in both data sets. We picked *extraTrees*, since it is a representative member of tree-based algorithms and is less affected by the increase in the number of its features in terms of runtime. The experiments were conducted by

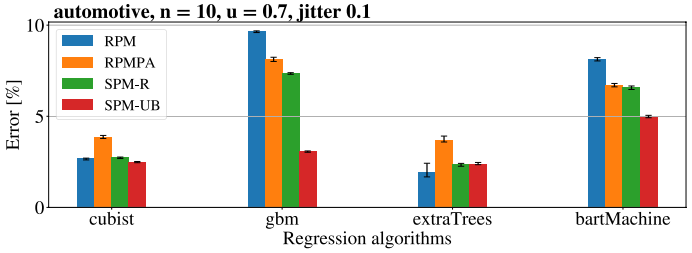

Fig. 9 Space-pruning for traces with jitter

generating 20 random splits of the data set into training and testing sets (for every parameter value). The model would then be fit on the training data and the average error measured on the test data.

Figure 7a shows how the error for *extraTrees* decreases when we include more features. The gain in accuracy becomes insignificant after adding more than three features from periodogram and autocorrelation. Thus, we kept three features from each of the periodogram and autocorrelation (i.e., six in total). We further analyzed the impact of the number of candidates for RPMPA method on accuracy. As shown in Fig. 7b, a relatively small number of candidates is required in order to achieve a low error until it reaches saturation.

6.3 Assessing accuracy in periodic systems

Impact of system utilization. Figure 8a and b show the average error as a function of the total utilization for task sets with 8 tasks. The error of the regression models increase with the increase in the utilization (which in-turn increases the number of preemptions). Furthermore, we observe a dramatic reduction in PeTaMi's accuracy when it is applied on log-uniform traces. This decrease is a result of having non-harmonic periods in log-uniform traces. In contrast, we see that the accuracy of our regression-based solutions has not been negatively affected when applied on non-harmonic periods.

Impact of the number of tasks. Figure 8c and d shows that the error reduces when there are more tasks in log-uniform traces for some of the tree-based solutions such as *gbm*. It is due to the decrease in the individual task utilization. Thus, although the system is as congested, the individual projection of a task contains larger idle intervals and shorter execution times that are likely not preempted much. This enables the periodogram to extract more meaningful features. However, in automotive task sets, the algorithms are rather unaffected by the number of tasks in the trace since they already have a good performance even for lower number of tasks.

Impact of execution time variations. From Fig. 8e and f, we observe that the regression-based methods are more robust to runtime execution-time variations than the baselines, showing a similar trend for both types of traces to the case with constant execution time. Also, with an increase in the execution time variation in Fig. 8g and h, we notice that most of the RBML methods are robust

(w.r.t. to this variation) for automotive traces, while for log-uniform traces, the error decreases with the increase in the execution-time variation. This behavior is due to the reduction in the average execution time for individual tasks. Since the execution time for a job is drawn from a uniform distribution in the range $[(1-\alpha)\times WCET, WCET]$, the wider the interval becomes the lower is the average execution time. Having smaller execution time is associated with lower utilization for the system and we previously observed that the methods perform better in lower utilization values.

Impact of release jitter. Figure 8i and j show that the release jitter has a much bigger impact on the error than the execution-time variation. One possible explanation is that the periodogram, which provides most of the information to the algorithms, is negatively impacted by jitter, thus, it produces less useful features for training. However, we observe that some regression algorithms such as *extraTrees* and *cubist* are still able to keep a low error even for this challenging scenario.

Impact of candidate adjustment method (RPMPA). Figure 8p and q show that RPMPA has about 50% less error than RPM for most RBML methods when used for cases with execution time variation and, implicitly, on ideal traces too (i.e., traces that do not have variations in the execution time or release time of the tasks). However, when the signal-processing techniques (periodogram and autocorrelation) are disturbed, as is the case for release jitter, the period adjustment step has a negative impact on the accuracy. Later in Table 4, we see a similar pattern for systems scheduled by non-preemptive scheduling policies.

Impact of space-pruning method (SPM). By analyzing Fig. 8r we observe that the inclusion of an upper and a lower bound for SPM contributes to reducing the error even further, proving that the regression is still prone to mistakes even when choosing candidates. However, this solution is expected to show little benefits for systems with large utilization, when fewer intervals of idle-time will be present. The reason is that in that case, SPM will provide upper bounds that are so large that they will not contribute much to filtering infeasible candidates. As we will show in Sect. 5.7, using quaternary projections can significantly reduce these period bounds for the SPM method.

While conducting this experiment, we noticed that under specific setups there can be cases where no candidates are left after the pruning phase. This situation occurs most frequently when the release time of the jobs is affected by jitter so much that the signal processing techniques only generate useless candidates that fall outside of the valid period bounds. As a consequence, when analyzing the effect of release jitter, we defined two criteria to provide a period estimate when no candidate is available. Namely, we either select the output of regression (SPM-R) or we select the upper bound (SPM-UB). Figure 9 shows the results for SPM on traces with jitter. In all cases, both versions of SPM succeed in reducing the error of RPMPA by 45% points. Also, SPM-UB is able to achieve an average error below RPM for *cubist*, *gbm*, and *bartMachine*, while for *extraTrees*, although it has a larger error, it presents a much narrower confidence interval. Thus, we can expect that the estimate of SPM-UB based on *extraTrees* to be more reliable than the corresponding RPM.

6.4 Assessing robustness

Next question to answer is how robust is our solution w.r.t. uncertainties in the underlying system that may drastically influence the traces generated from those systems. In the rest of this section, we evaluate the robustness of our solution in the presence of (i) higher-priority aperiodic tasks (Sect. 5.4.1), (ii) dropped or discarded jobs (Sect. 5.4.2), (iii) overloads (Sect. 5.4.3), and (iv) initial offsets (Sect. 5.4.4) in the system.

6.4.1 Robustness w.r.t. the presence of higher-priority aperiodic tasks

For this experiment, we considered a configuration consisting of 12 automotive tasks (6 periodic and 6 sporadic tasks) scheduled by Rate Monotonic scheduling policy that are interfered by high-priority aperiodic tasks arriving according to a Poisson process with a rate $\lambda = 0.0005$ events/ns (namely, roughly 5 arrivals in every 10us). Furthermore, we focused on analyzing one periodic task in scenarios of having high, medium, and low priority, respectively. For each of the three priority scenarios, the task's priority has been chosen randomly in the ranges [1, 3] for high, [4, 7] for medium, and [8, 12] for low-priority tasks.

Figure 8m to o show the error as a function of utilization for the tree-based algorithms, periodogram and PeTaMi. The periodogram is affected significantly when priority changes from high to low (comparing Fig. 8m and o) due to the increase in the number of preemptions in low-priority tasks which in-turn causes more noise in periodogram. In contrast, the error of RPM algorithms increases only slightly in large utilization values. We also see that the error of *gbm* and *bartMachine* at low utilization values is smaller when the task under analysis has a low priority. It is due to the fact that these two algorithms may not be able to generalize well when the periodogram has low error. Having a low error for periodogram means having less significant (shorter) peaks, which in turn do not provide enough information for these algorithms to excel.

6.4.2 Robustness w.r.t. dropping jobs

Next, we explore the impact of having missed (dropped) jobs in the input projections on the accuracy of our solutions. The setup includes 10 automotive tasks. We consider two scenarios: (i) the tasks under analysis has dropped jobs (with a 15% probability), and (ii) all the other tasks have dropped jobs. Figure 8k and l show that all algorithms exhibit a relatively higher error when there are dropped jobs and the utilization is higher in comparison with experiments with no dropped job (e.g., comparing Fig. 8a and k or l). This increase is due to the fact that projections are imperfect and even can be misleading when some jobs are dropped. Moreover, periodogram is affected notably when the task under analysis drops jobs. However, while the RBML methods show little variations from one case to

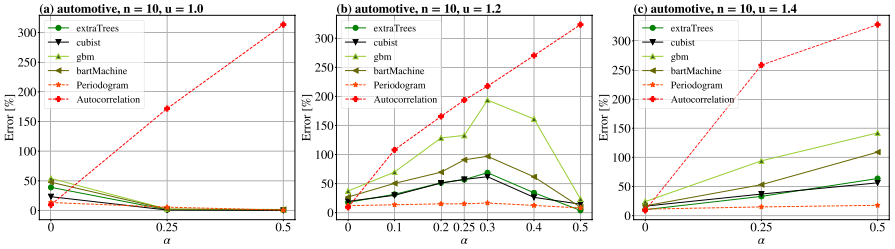

Fig. 10 The impact of permanent overloads

the other, they are still able to retain meaningful information from their features even when the task under analysis has a low utilization.

6.4.3 Robustness w.r.t. permanent overloads

For this case, we experimented with task sets whose total utilization exceeded 100%. A consequence of such an overload is that lower-priority tasks will experience *starvation* (i.e., these tasks will not get any opportunity to be executed). Since the projections of starved tasks do not include any information about their periodicity, we excluded such tasks from both the training and testing phase.

Figure 10 illustrates the effect of tardiness on the performance of the four tree-based regression algorithms and on the two signal processing techniques as a function of the execution time variation. In Fig. 10a we observe a decrease in the error with respect to the execution-time variation factor α when the total utilization of the system is 100%. This trend is due to the decrease in the average utilization with the increase in α. For instance, for $\alpha = 0.5$, the execution time of the tasks will be uniformly drawn from $[0.5 \times \text{WCET}, \text{WCET}]$, which implies an average execution of $0.75 \times \text{WCET}$, hence, the total utilization will be around 75% rather than 100%.

However, in Fig. 10c we observe an opposite trend for 140% utilization. In this case, the smaller values of error when there is no execution time variation are due to the elimination of the tasks suffering from starvation (as mentioned earlier, they disappear from the trace).

On the other hand, we see an increase in the error for larger α values. It is due to the large execution time variations which then allows some of the previously starved low-priority tasks find chances to be executed. However, since these chances appear randomly and infrequently, the resulting projections would not be consistent enough to provide meaningful data for the regression algorithms both during the training phase and testing phase.

Figure 10b reflects the combination of the observations for the previous two cases. Until 30% execution time variation, the total utilization does not fall under 100%, hence Fig. 10b shows a similar trend to Fig. 10c. However, when α increases, the curve becomes similar to Fig. 10a, since now the system allows more systematic running intervals for the lower-priority tasks.

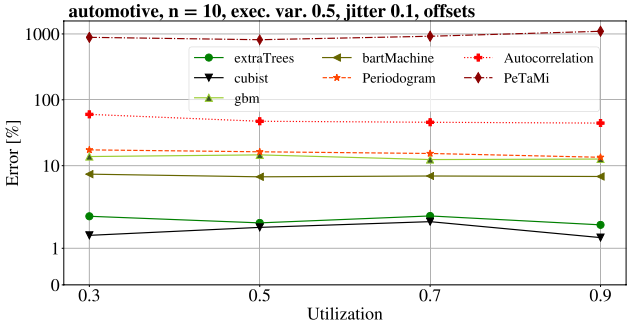

Fig. 11 Impact of offsets

Table 2 Results on the two CAN data sets

Data set	Algorithm	RPM (%)	RPMPA (%)
CAN 1 (Lee et al. 2017; HCRL 2010)	*extraTrees*	28.2568	1.6179
	cubist	**3.2461**	**0.9703**
	gbm	15.8224	1.3919
	bartMachine	20.8588	14.0103
	Periodogram	5.3368	–
CAN 2 (Seo et al. 2018; HCRL 2010)	*extraTrees*	13.0256	**1.7039**
	cubist	**5.5005**	2.8963
	gbm	19.8341	9.8881
	bartMachine	14.956	5.0264
	Periodogram	9.5448	–

Bold values indicate the smallest error in estimating message periods in each of the two datasets by the PRM and PRMPA methods

6.4.4 Robustness w.r.t. offsets

So far, the majority of our experiments were focused on synchronous tasks (namely, the initial offset of the tasks was zero). However, there are many scenarios where the presence of offsets is an essential element of the system, e.g., when the trace has been gathered from messages that transfer over a controller area network (CAN) and are generated by unsynchronized nodes.

We performed an experiment similar to the one for periodic systems (Sect. 5.3) where we varied the utilization of the task set. For this experiment, we used automotive task sets with 10 periodic tasks, 50% execution time variation, and 10% release jitter. The offsets were randomly selected from the range [0, H/2] with a uniform distribution, where H is the hyperperiod of the tasks. The results are presented in Fig. 11.

The first observation in Fig. 11 is that most of the RBML methods have an almost constant error regardless of the utilization. This behavior is due to the decrease in the average utilization, since a variation of 50% in execution implies

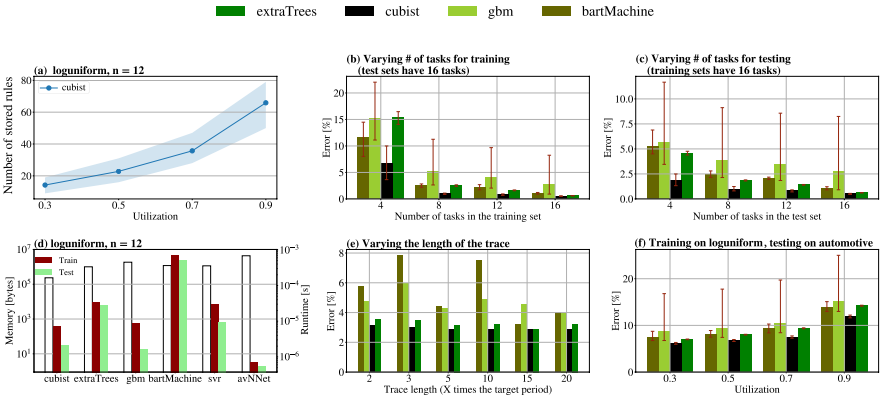

Fig. 12 Other evaluation criteria: **a** the number of rules generated by *cubist* algorithm as a function of utilization, **b**, **c** the impact of training and testing on data sets with different number of tasks, **d** the memory consumption and runtime of the RBML methods, **e** the impact of the trace length on the accuracy, and **f** the impact of training and testing on different task set types on the accuracy

an average execution time 75% × WCET, which in turn makes our actual utilization values to be small enough for the RBML methods to perform similarly. Also, we notice both *cubist* and *extraTrees* having an error below 5% for all utilization values, while PeTaMi shows errors reaching to 1000%. We have seen a similar poor performance from PeTaMi in Fig. 8i for task sets with release jitter (without offset). It seems that the addition of offsets and execution time variation appears to exacerbate the impact of release jitter when considering different utilization values.

6.5 Case study

In this section, we validate our period-inference methods on two case studies. We use two data-sets consisting of traces coming from controller area networks (CANs), denoted by CAN 1 (Lee et al. 2017; HCRL 2010) and CAN 2 (Seo et al. 2018; HCRL 2010) in Table 2. The first data set consists of 988,987 messages with 27 tasks and the second one of 2,369,868 messages with 45 tasks.

In order to generate our test data, we split the projections from the messages into smaller projections of 100 jobs. As for our training data, we synthetically generated traces that would provide a good proxy for real data, namely, we created a data set of 6000 automotive traces, with 20 tasks scheduled by non-preemptive rate monotonic scheduling policy, with 50% utilization and 5% jitter as the training set. The results from Table 2 show that our methods successfully estimated the periods of the messages on the actual use case, having errors below 2% for both data sets.

6.6 Assessing other aspects

6.6.1 Number of rules for *cubist*

Figure 12a shows an increase in the number of rules generated by *cubist* as the utilization grows for log-uniform traces with 12 tasks. This behavior is expected since the projections become more complex as the number of preemptions increases in higher utilization values. For example, the average number of rules stored at 30% utilization is 16, but it raises to 65 at 90% utilization.

6.6.2 Learning robustness: training and testing on different number of tasks

Figure 12b and c illustrate how the mismatch between the number of tasks within the traces used for training and the traces used for testing affects the accuracy of the tree-based algorithms. The experiment is conducted on log-uniform traces with 70% utilization and without uncertainties (since we want to capture a large range of periods and also isolate the effect of the discrepancy between the number of tasks).

Figure 12b shows that training on traces with fewer tasks than the target system leads to 2.9% higher error in average than the opposite scenario (Fig. 12c). Moreover, in both cases, we observe that when the gap between the number of tasks in training and testing traces is smaller, the error is smaller too. We also see that *cubist* has the best generalization capability; it has less than 2.5% error as long as it is trained with task sets that have at least 8 tasks.

6.6.3 Learning robustness: training and testing on different task set types

For this experiment, we train the models on an *aggregated set* containing 2000 log-uniform traces for every utilization value in {0.3, 0.5, 0.7, 0.9} and 12 tasks. Afterwards, the evaluation is completed on 2000 automotive traces with 12 tasks for each of the aforementioned utilization values *individually*.

The results, summarized in Fig. 12f, show that *cubist* has the smallest error (i.e., below 8% for utilization values lower than 90%). Comparing Figs. 12f and 8a (where the training and testing were done on the same task set types) we see only a slight difference in the error of the RBML methods which shows that they rather generalize well.

6.6.4 Learning robustness: training and testing on different projection lengths

The goal of this experiment is to see how the error of the RBML methods is affected by the length of their input trace during the inference phase (testing). This experiment is performed on log-uniform traces with 70% utilization and 10 tasks, with the particularity that, when testing, we limit the length of the projections to *a certain multiple of the task's period* (shown on the horizontal axis of Fig. 12e). As it can be seen in Fig. 12e, both *cubist* and *extraTrees* are able to estimate the true period with less than 4% error even when only two jobs of the target task appear in the trace. As expected, the error reduces gradually with the increase in the length of the trace. In contrast, *bartMachine*

Table 3 Evaluating the learning robustness when the training and testing are done with different scheduling policies

Algorithm	Utilization	RM (%)	EDF (%)
cubist	0.3	0.1047	0.1046
	0.5	0.1122	0.1057
	0.7	0.1552	0.1554
	0.9	0.2165	0.2024
gbm	0.3	0.7145	0.7151
	0.5	0.8730	0.8731
	0.7	0.9558	0.9493
	0.9	1.0915	1.0951
extraTrees	0.3	0.3024	0.3026
	0.5	0.3951	0.3953
	0.7	0.4891	0.4821
	0.9	0.8702	0.9175
bartMachine	0.3	0.5947	0.5929
	0.5	0.6332	0.6316
	0.7	0.9501	0.9509
	0.9	1.3092	1.3155

The columns indicate the policy being used for testing

has the largest fluctuations in error, making it less reliable when the projection's length is lower than 20 times the period of the target task.

6.6.5 Learning robustness: training and testing on task sets scheduled by different scheduling policies

One other aspect we take into account is the robustness of our method when there could be a difference between the scheduling policy used during the training and testing phases. In this experiment, we train the RBML methods on traces of task sets coming from both rate monotonic (RM) and earlier-deadline first (EDF) scheduling policies. Afterwards, we test the RBML models on separate test sets, each containing traces only from RM and EDF, respectively. Also, since the periods coming from a log-uniform distribution are not harmonic, we chose log-uniform traces for this experiment. Since for harmonic periods (as in automotive traces), RM generates an almost identical schedule to EDF, it is not useful to consider those task sets in this experiment.

For this experiment, task sets where generated following the same approach explained in Sect. 5.3. Each task set has 10 tasks with log-uniform periods and 50% execution time variation. Table 3 shows that all considered regression methods have a very similar performance regardless of the utilization.

6.7 Quaternary projections

As mentioned in Sect. 4.2, the information about the execution of lower-priority tasks in quaternary projections can be used to reduce the upper bound on the

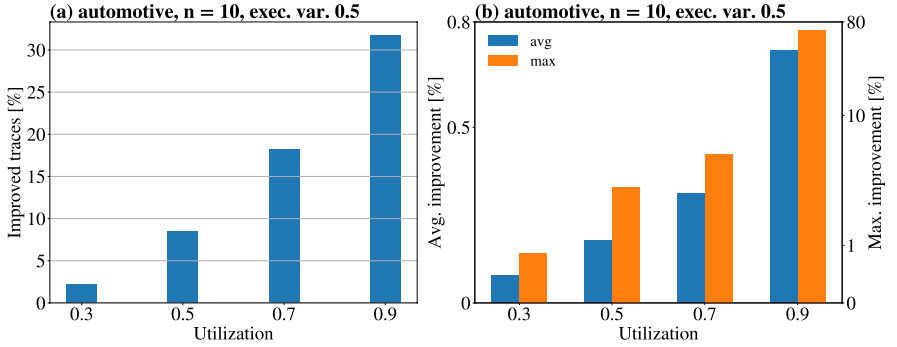

Fig. 13 The effect of quaternary projections. **a** Is the percentage of the traces that had an improvement in their bounds when quaternary projections were used and **b** is the average improvement (left vertical axis) and maximum improvement (right vertical axis). Please note the difference in the scale of these two vertical axes

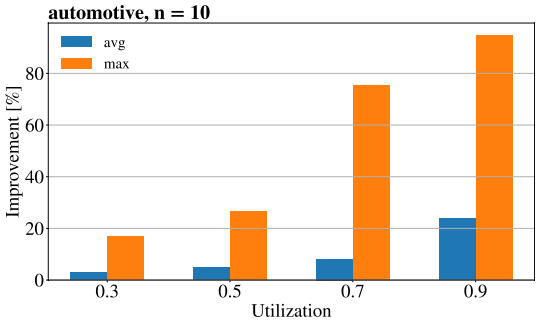

Fig. 14 Comparing Theorem 1 with Theorem 1 in Vădineanu and Nasri (2020) to quantify the improvement in the new upper bound

period. To evaluate the impact of this extra information in reducing the period bounds, we perform an experiment to compare the accuracy improvement resulted from using bounds obtained from quaternary and ternary projections.

Our metric is the percentage of improvement in the bounds (the smaller the upper bound, the better it is). We reported $(B^T - B^Q)/B^T$, where B^Q is the upper bound resulted from quaternary projections and B^T is the upper bound resulted from ternary projections (see Sects. 4.1 and 4.2). A higher value of improvement indicates that the upper bound obtained from the quaternary projection is smaller (tighter) than the upper bound from the ternary projection. The experiment configuration is similar to those described in Sect. 5.1. We consider automotive tasks with 10 tasks each having a 50% execution time variation and no release jitter.

Figure 13a shows the percentage of the traces that have an improvement in their bounds when quaternary projections are used, i.e., when $B^Q < B^T$. Figure 13b shows the average improvement (left vertical axis) and maximum improvement (right vertical axis). These average and maximum improvement values are obtained for projections for which using quaternary projections improves the upper bound.

Figure 13 shows that with the increase in the utilization, the improvement on the bounds also increases (both the percentage of task sets that have improvement in their bound and the amount of improvements). This is expected since at larger utilization values, the idle intervals become more scarce and, therefore, less informative. However, the intervals during which lower-priority tasks execute either do not change or increase (like all other tasks when a system has higher utilization). Consequently, the quaternary projections become richer and richer in terms of information they contain.

6.8 New upper bound

In this experiment, we evaluate the improvement resulted from the new upper bound (Theorem 1 in this paper) and the upper bound in our prior work [Theorem 1 in (Vădineanu and Nasri 2020)]. We report this improvement by $(B^{old} - B^{new})/B^{old}$, where B^{new} is the upper bound from Theorem 1 (in this paper) and B^{old} is the upper bound from Theorem 1 in Vădineanu and Nasri (2020).

We performed the experiment as explained in Sect. 5.1. We varied the utilization and for each utilization value, the experiment was performed on 2000 automotive traces with 10 tasks each without any uncertainties in their timing parameters (i.e., no release jitter).

Figure 14 shows the average and maximum improvements for all projections generated in the experiment. We observe that with the increase in the utilization, the new bound becomes tighter than the old bound (the improvement increases). As we discussed earlier, with the increase in the utilization, the chance to find idle times in a ternary projection reduces and hence it becomes more important to use the remaining opportunities (resulted from the few remaining idle slots) more efficiently. The new bound uses these opportunities more efficiently by having a tighter estimation of the arrival time of the second effective interval in every two consecutive effective intervals. This can also be seen when comparing Eq. (12) in this paper and Eq. (10) in Vădineanu and Nasri (2020).

6.9 Non-preemptive scheduling

To evaluate performance of our solution for systems that are scheduled by a non-preemptive scheduling policy, we performed a similar set of experiments as in Sect. 5.3, using automotive task sets for the following utilization values {0.15, 0.3, 0.45, 0.6}. We scheduled these task sets by the non-preemptive fixed-priority scheduling policy (following rate monotonic priorities). The reason we could not include higher utilization values was due to the large amount of deadline misses that appear in traces. Since we wanted to focus on the impact of the non-preemptive scheduling policies in the experiments (and not the impact of deadline misses), we excluded higher utilization values from this experiment.

We excluded *svr* and *avNNet* from the diagrams because they had a poor performance (similar to Fig. 8 for the non-preemptive systems as well. We also

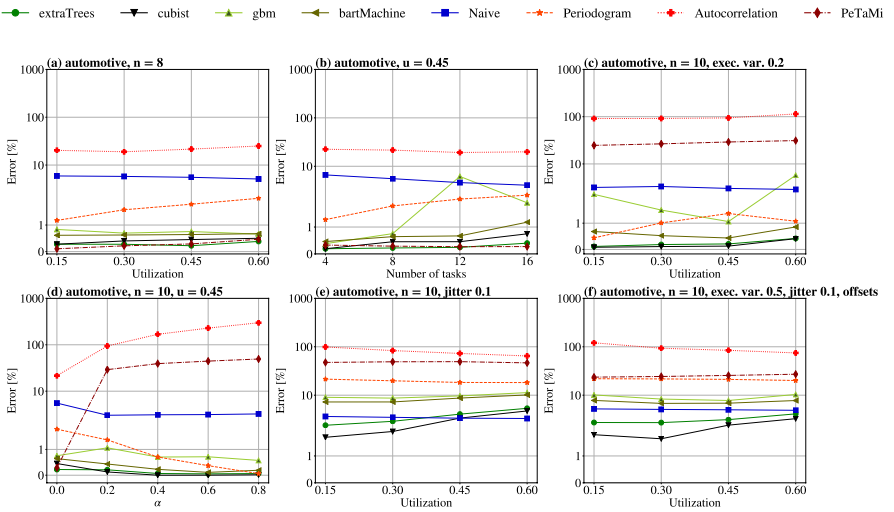

Fig. 15 Results for non-preemptive systems. **a, b** The impact of utilization and the number of tasks on task sets with no timing uncertainties, **c** the impact of utilization on tasks with execution time variation, **d** the impact of execution time variation, **e** the impact of utilization on tasks with release jitter, and **f** the impact of utilization on tasks with release jitter and random offset

Table 4 Comparing the accuracy improvement resulted from RPMPA and SPM w.r.t. the original RPM for non-preemptive systems considered in Fig. 15. Negative improvements represent a drop in accuracy

Algorithm	Setup	RPMPA improvement (%)	SPM improvement (%)
cubist	No uncertainty	8.16	90.09
extraTrees		13.36	86.44
bartMachine		25.75	89.56
gbm		28.59	87.81
cubist	Execution var.	11.37	86.31
extraTrees		53.55	87.49
bartMachine		41.29	86.25
gbm		42.93	91.41
cubist	Release jitter	− 60.75	22.28
extraTrees		− 15.01	39.87
bartMachine		22.10	69.09
gbm		27.02	72.86

added one more baseline, called the *naive* solution that calculates the *mean of the inter-arrival intervals within the projection* as an estimation for the period of the task.

Figure 15a shows how the average error is influenced by the change in utilization. We notice a slight increase in error for every tree-based algorithm with the increase in utilization (this is similar to the preemptive systems). Also, the errors of

all RBML methods stay below 1% with *cubist*, *extraTrees*, and PeTaMi producing very similar results.

Impact of the number of tasks. Figure 15b shows that the error increases with the increase in the number of tasks, which is the opposite of the trend we saw in Fig. 8c. This increase is caused by the increase in the blocking of higher-priority tasks. In a preemptive system, higher-priority tasks typically are scheduled as soon as they are released, but in a non-preemptive system, they might be blocked by the lower-priority tasks. With the increase in the number of tasks, the chance that a higher-priority task is being blocked by a low-priority one increases.

Impact of execution-time variation. Figure 15c shows that adding even a small amount of execution-time variation significantly increases the error of PeTaMi. We see a slight increase (of 5%) in the error of *gbm*, while *cubist*, *extraTrees* and *bartMachine* are largely unaffected. Moreover, we observe from Fig. 15d that increasing the execution time variation (the horizontal axis) results in a decrease of the average error. This behavior is similar to what we saw in preemptive systems (shown in Fig. 8g and explained in Sect. 5.3).

Impact of release jitter. Similar to the preemptive case (see Sect. 5.3 the part on release jitter), having release jitter in the system increases the error for all methods, in particular for PeTaMi. As we expected, when there are inherent uncertainties in the arrival times (caused by release jitters), the accuracy of PeTaMi decreases drastically, while our solutions have a low error (below 7%).

Impact of initial task offsets. To assess the impact of initial offsets on the accuracy of our solutions for non-preemptive systems, we performed another experiment in which we assigned a random offset with a uniform distribution in the range $[0, H/2]$ to each task, where H is the hyperperiod of the task set. Fig. 15f shows the impact of random offsets (its offset-free counterpart is Fig. 15e). We observe that adding offsets hardly changes the behavior of most of the methods since Fig. 15f looks very similar to Fig. 15e. Interestingly, adding offsets improved PeTaMi's accuracy (by about 20%). One explanation might be that when a set of almost harmonic tasks have non-identical offsets, the chance that they arrive at a time that the resource is idle is higher. Consequently, when they arrive, they are not interfered or being blocked by other higher- or lower-priority tasks. Therefore, the uncertainty about the execution window of the tasks, and hence the error of PeTaMi, reduces.

Comparing RPMPA and SPM with RPM. This experiment quantifies the degree of improvement introduced by the period adjustment method (RPMPA) and by the space-pruning method (SPM) in systems that consist of non-preemptive tasks. For this experiment, we aggregated the task sets generated for Fig. 15 into three categories: (i) task sets with no execution-time variation and no jitter (including all task sets used for Fig. 15a and b), (ii) task sets with execution-time variation (including all task sets used for Fig. 15c and d), (iii) task sets with release jitter (including all task sets used for Fig. 15e and f). These categories are represented in the second column of Table 4 with labels *no uncertainty*, *execution var.*, and *release jitter*. We then run SPM and RPMPA on each of the categories and compare their performance with RPM.

To evaluate the impact of RPMPA and SPM w.r.t. RPM, we used a metric called *improvement %* shown on the third and fourth columns of Table 4. This metric is calculated as follows $(E^{RPM} - E^{X})/E^{RPM}$, where X is either RPMPA or SPM. This

metric shows by what percentage each of the RPMPA and SPM could further reduce the error of the RPM method. The higher the value of the improvement, the larger the positive impact of the method. When the improvement is negative, the method being considered has increased the error in comparison to the original RPM. This case happened, for example, for RPMPA when task sets have release jitter (as it can be seen in the third column of Table. 4 for *cubist* and *extraTrees* methods).

Table 4 shows that applying period bounds (i.e., the SPM method) significantly reduces the error in comparison to the original RPM. For task sets that do not have uncertainties or have execution-time variation, the average improvement is about 85% (see the upper part of the fourth column of Table 4). For task sets with release jitter, the benefit is a bit smaller but it is certainly more than just using RPMPA (comparing the third and fourth columns). In average, SPM has an 80% improvement in the accuracy (for all task sets in Fig. 15 combined).

In particular, for systems with release jitter, using the SPM certainly (and always) reduces the errors while this might not be the case for RPMPA. As it was discussed earlier (in Sect. 5.3), release jitter negatively impact the signal-processing techniques (periodogram and autocorrelation) and hence the candidates they produce may be farther from the true period.

7 Discussions

ExtraTrees and cubist. Among our experiments, *extraTrees* algorithm has the lowest error in almost all setups. However, when it trains on a different task set type than the ones used for testing (shown in Sects. 5.5 or 5.6.3), its does not generalize well. This makes *extraTrees* a good choice when we have access to training data from the same system we want to test on (e.g., when it is used for monitoring a known system). On the other hand, when the target system is unknown, or we do not have access to the traces, *cubist* regression is a better choice; with a similar accuracy as *extraTrees* and better generalizability.

Non-tree-based solutions. The experiments have confirmed our statement from Sect. 3.2, where we expected the tree-based solutions to outperform the other types of RBML methods, when applied to our problem. We noticed that *avNNet* has almost a constant yet very large error of 100% for all experiments. This expresses the inability of the algorithm to learn a non-linear mapping from the input to the output, deciding instead to approximate the output as a constant value, namely the average of the periods from the training set. Since the test set is generated the same way as the training set, it will have a similar average value for its periods, thus keeps the error constant. Furthermore, *svr* is also not a good choice for the period-inference problem since our feature space is rather sparse, namely the features from periodogram and autocorrelation do not have values that place the data points close to each other in this space. Hence, *svr* is not able to find a suitable hyperplane to fit the data. Moreover, we see that the candidate adjustment step has a significant impact on reducing the errors. However, in the presence of large release jitters, RPMPA must be used cautiously as it may increase the error.

Discussions on memory consumption and runtime. Figure 12d addresses both the memory requirements and the runtime of the six considered RBML methods. We notice that *cubist* has a considerably low memory consumption compared to the rest. It is also almost the fastest solution during both the training and testing phases among the well-performing algorithms. On the other end of the spectrum, *bartMachine* has the slowest training and testing phases, and *avNNet* has the largest memory consumption among the considered algorithms.

Non-preemptive scheduling. By conducting a set of experiments for systems without preemptions (see Fig. 15), we observed that the behavior expressed by all tree-based solutions was highly similar to the preemptive case. This demonstrates the applicability of our solutions to both types of scheduling paradigms.

Offsets. We noticed that the addition of offsets hardly resulted in a change of behavior for the RBML methods, presenting an almost identical pattern to the scenario with simultaneous releases.

Comparing RPM, RPMPA, and SPM. Our results (see Fig. 9; Tables 2 and 4) show that both RPMPA and SMP are able to further reduce the error resulted from the regression methods (RPM method). We also observed a significant reduction of error when using period bounds (SPM method) in comparison with RPM.

8 Conclusions

In this paper, we introduced the first regression-based machine learning (RBML) solution for the problem of inferring a task's period from its binary projections. We investigated six most-successful families of RBML methods for this problem and provided comprehensive evaluations and discussions about their accuracy and robustness under various scenarios. We proposed further steps for improving the accuracy by creating period-adjustment and space-pruning methods that use the properties of a work-considering scheduler to prune the space of valid periods of a task. Our solutions proved to be robust and highly accurate. The average observed error of our (best) solution was under 1% in most scenarios including those with a mixture of periodic, aperiodic, and sporadic tasks, execution time variation, and release jitter while the existing work has two to three orders-of-magnitude higher errors. On the case studies from actual systems, the error of our best solution was 1.7%. In the future, we would like to explore RBML methods to infer the timing properties of parallel applications running on multiprocessor platforms and under partial observations.

Acknowledgements We would like to thank your reviewers for their constructive feedback. This work was carried out on the Dutch national e-infrastructure with the support of SURF Cooperative.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

Akesson B, Nasri M, Nelissen G, Altmeyer S, Davis RI (2020) An empirical survey-based study into industry practice in real-time systems. In: IEEE real-time systems symposium (RTSS). IEEE, Houston, pp 1–9

Audsley N, Burns A, Richardson M, Tindell K, Wellings AJ (1993) Applying new scheduling theory to static priority preemptive scheduling. Softw Eng J 8(5):284–292

Barr ET, Harman M, McMinn P, Shahbaz M, Yoo S (2015) The oracle problem in software testing: a survey. IEEE Trans Software Eng 41(5):507–525

Bellman R (1961) Curse of dimensionality. In: Adaptive control processes: a guided tour. Princeton University Press, Princeton, p 2

Berberidis C, Aref WG, Atallah M, Vlahavas I, Elmagarmid AK (2002) Multiple and partial periodicity mining in time series databases. In: European conference on artificial intelligence (ECAI). ECAI, Vienna, pp 370–374

Breiman L, Friedman J, Stone CJ, Olshen RA (1984) Classification and regression trees. CRC Press, Boca Raton

Chéramy M, Hladik PE, Déplanche AM (2014) Simso: a simulation tool to evaluate real-time multiprocessor scheduling algorithms. In: International workshop on analysis tools and methodologies for embedded and real-time systems (WATERS). HAL, Madrid

Chipman HA, George EI, McCulloch RE et al (2010) BART: Bayesian additive regression trees. Ann Appl Stat 4(1):266–298

Cortes C, Vapnik V (1995) Support-vector networks. Mach Learn 20(3):273–297

Delgado MF, Sirsat MS, Cernadas E, Alawadi S, Barro S, Febrero-Bande M (2019) An extensive experimental survey of regression methods. Neural Netw 111:11–34

Emberson P, Stafford R, Davis RI (2010) Techniques for the synthesis of multiprocessor tasksets. In: International workshop on analysis tools and methodologies for embedded and real-time systems (WATERS), pp 6–11

Friedman JH (2002) Stochastic gradient boosting. Comput Stat Data Anal 38(4):367–378

Geurts P, Ernst D, Wehenkel L (2006) Extremely randomized trees. Mach Learn 63(1):3–42

Greenwell B, Boehmke B, Cunningham J, Developers G (2019) GBM: generalized boosted regression models. Available at https://CRAN.R-project.org/package=gbm. R package version 2.1.5

Gubner JA (2006) Probability and random processes for electrical and computer engineers. Cambridge University Press, Cambridge

HCRL (2010) http://ocslab.hksecurity.net/Datasets. Accessed on 21 Oct 2020

Iegorov O, Fischmeister S (2018) Mining task precedence graphs from real-time embedded system traces. In: IEEE real-time and embedded technology and applications symposium (RTAS). IEEE, Porto, pp 251–260

Iegorov O, Torres R, Fischmeister S (2017) Periodic task mining in embedded system traces. In: IEEE real-time and embedded technology and applications symposium (RTAS). IEEE, Pittsburgh, pp 331–340

Kapelner A, Bleich J (2016) bartMachine: machine learning with Bayesian additive regression trees. J Stat Softw 70(4):1–40. https://doi.org/10.18637/jss.v070.i04

Kramer S, Ziegenbein D, Hamann A (2015) Real world automotive benchmarks for free. In: International workshop on analysis tools and methodologies for embedded and real-time systems (WATERS)

Kuhn M (2020) caret: classification and regression training. Available at https://CRAN.R-project.org/package=caret. R package version 6.0-86

Kuhn M, Quinlan R (2020) Cubist: rule- and instance-based regression modeling. Available at https://CRAN.R-project.org/package=Cubist. R package version 0.2.3

Lamichhane K, Moreno C, Fischmeister S (2018) Non-intrusive program tracing of non-preemptive multitasking systems using power consumption. In: Design, automation & test in Europe conference & exhibition (DATE). IEEE, Dresden, pp 1147–1150

Lee H, Jeong SH, Kim HK (2017) OTIDS: a novel intrusion detection system for in-vehicle network by using remote frame. In: Annual conference on privacy, security and trust (PST). IEEE, Calgary, pp 57–5709

Leondes CT (1996) Computer techniques and algorithms in digital signal processing: advances in theory and applications. Elsevier, Amsterdam

Li TH (2012) Detection and estimation of hidden periodicity in asymmetric noise by using quantile periodogram. In: IEEE international conference on acoustics, speech and signal processing (ICASSP). IEEE, Kyoto, pp 3969–3972

Malode Y, Khadse D, Jamthe D (2015) Efficient periodicity mining using circular autocorrelation in time series data. Int Res J Eng Technol 2(3):430–436

McKilliam RG, Clarkson IVL, Quinn BG (2014) Fast sparse period estimation. IEEE Signal Process Lett 22(1):62–66

Meyer D, Dimitriadou E, Hornik K, Weingessel A, Leisch F (2019) e1071: misc functions of the Department of Statistics, Probability Theory Group (Formerly: E1071), TU Wien. https://CRAN.R-project.org/package=e1071, r package version 1.7-3

Nasri M, Chantem T, Bloom G, Gerdes RM (2019) On the pitfalls and vulnerabilities of schedule randomization against schedule-based attacks. In: IEEE real-time and embedded technology and applications symposium (RTAS). IEEE, Montreal, pp 103–116

Puech T, Boussard M, D'Amato A, Millerand G (2019) A fully automated periodicity detection in time series. In: International workshop on advanced analysis and learning on temporal data (AALTD). Springer, Cham, pp 43–54

Quinlan J (1992) Learning with continuous classes. In: Australian joint conference on artificial intelligence (AJCAI), vol 92. World Scientific, Singapore, pp 343–348

Quinlan J (1993) Combining instance-based and model-based learning. In: International conference on machine learning (ICML). ICML, Sydney, pp 236–243

Quinlan J (2014) C4.5: programs for machine learning. Elsevier, Amsterdam

Ripley BD (2007) Pattern recognition and neural networks. Cambridge University Press, Cambridge

Salem M, Crowley M, Fischmeister S (2016) Anomaly detection using inter-arrival curves for real-time systems. In: Euromicro conference on real-time systems (ECRTS). IEEE, Toulouse, pp 97–106

Schuster A (1898) On the investigation of hidden periodicities with application to a supposed 26 day period of meteorological phenomena. Terr Magn 3(1):13–41

Seo E, Song HM, Kim HK (2018) GIDS: GAN based intrusion detection system for in-vehicle network. In: Annual conference on privacy, security and trust (PST). IEEE, Belfast, pp 1–6

Simm J, de Abril IM, Sugiyama M (2014) Tree-based ensemble multi-task learning method for classification and regression. http://CRAN.R-project.org/package=extraTrees

Sucholutsky I, Narayan A, Schonlau M, Fischmeister S (2019) Deep learning for system trace restoration. In: International joint conference on neural networks (IJCNN). IEEE, Budapest, pp 1–8

Unnikrishnan P, Jothiprakash V (2018) Daily rainfall forecasting for one year in a single run using singular spectrum analysis. J Hydrol 561(1):609–621

Vlachos M, Yu P, Castelli V (2005) On periodicity detection and structural periodic similarity. In: SIAM international conference on data mining. SIAM, Philadelphia, pp 449–460

Vădineanu S (2020) Regression-based period miner. Available at https://github.com/SerbanVadineanu/period_inference

Vădineanu S, Nasri M (2020) Robust and accurate period inference using regression-based techniques. In: IEEE real-time systems symposium (RTSS). IEEE, Houston, pp 358–370

Young C, Olufowobi H, Bloom G, Zambreno J (2019) Automotive intrusion detection based on constant can message frequencies across vehicle driving modes. In: ACM workshop on automotive cybersecurity. ACM, New York, pp 9–14

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Şerban Vădineanu is a Ph.D. candidate at Leiden Institute of Advanced Computer Science from Leiden University, the Netherlands. He received his B.Sc. degree in applied electronics from University "Politehnica" of Bucharest, Romania, the M.Sc. degree in embedded systems from Delft University of Technology, the Netherlands. This work is the result of his master thesis supervised by Dr. Mitra Nasri. His research interests include machine learning and deep learning, with particular interest in computer vision tasks with expensive data acquisition.

Mitra Nasri is an Assistant Professor at Eindhoven University of Technology, the Netherlands. She received her PhD from the University of Tehran, in 2015. Before that, she was an assistant professor at Delft University of Technology, the Netherlands, a postdoc fellow at the Max Planck Institute for Software Systems, Germany, and a postdoc researcher at TU-Kaiserslautern, Germany. Her research interests include scheduling, timing analysis, and model-based design of real-time systems.