Dynamic Scheduling for Event-Driven Embedded Industrial Applications

Hossein Taji*, José Miranda*, Miguel Peón-Quirós†, Szabolcs Balasi‡, and David Atienza*
*Embedded Systems Laboratory (ESL), EPFL, Switzerland, †EcoCloud, EPFL, Switzerland, ‡Nespresso, Switzerland
Email: *†,{hossein.taji, jose.mirandacalero, miguel.peon, david.atienza}@epfl.ch, ‡Szabolcs.Balasi@rd.nestle.com

***Abstract*—This paper addresses the optimization of embedded platforms to meet the computing and real-time requirements of cyber-physical systems and IoT applications, including embedded intelligence. In this context, schedulers are vital in enhancing processor utilization in industrial contexts. Although existing research has focused primarily on the schedulability of periodic tasks, event-driven tasks better represent these new embedded intelligence scenarios in the real world. This work explores static and dynamic scheduling policies within a general scenario and a specific case study based on an actual industrial application. The proposed dynamic scheduler has been integrated into the FreeRTOS kernel and has been employed to conduct all of our experiments on industrial products within the smart home domain. Our results show that, while we can respect real-time requirements, our proposed dynamic scheduling can improve the performance of event-driven applications by reducing missed task deadlines by up to $60\,\%$. Moreover, we have also developed a lightweight version of our dynamic scheduler for industrial products that reduces average timing overhead for task selection and insertion by up to $34.7\,\%$ and memory overhead for task creation and list scheduling by up to $74.7\,\%$ compared to state-of-the-art static alternatives.**

***Index Terms*—Event-Driven Tasks, Scheduling, RTOS, Embedded Systems, Industrial Applications**

I. Introduction

Cyber-physical systems (CPS) are evolving into increasingly intelligent entities as they embrace a multitude of advanced functionalities such as machine learning. However, the integration of these functionalities also results in higher processing demands. As a result, it is crucial to fully leverage the capabilities of the embedded platforms used in these systems. This is particularly important for CPS deployed in industrial products, especially considering the growing adoption of intelligence by industrial companies in their product offerings [1]. In fact, optimizing the integrated embedded platforms in industrial products can result in significant cost savings for mass production budgets.

In CPS with real-time constraints, the scheduler manages and synchronizes all tasks, guaranteeing the different time requirements [2]. Generally, schedulers are categorized into two groups: static and dynamic. The former is based on priority-assigned tasks. These priorities are established during system design and remain fixed, leading to consistent scheduling decisions during operation. While determining these priorities, especially when managing numerous tasks, is challenging and their optimality can be constrained in specific scenarios [3]–[5], even more significant is the struggle faced by static schedulers in handling the dynamic nature of applications where tasks are triggered by events. In contrast, a dynamic scheduler makes such a decision based on parameters computed during run-time. Previous studies have demonstrated that dynamic scheduling techniques tend to outperform static ones for a set of periodic tasks when dealing with non-overloaded task sets [3].

When discussing real-time systems in the state of the art, the focus has been on handling periodic tasks [3]–[8]. However, in CPS and especially industrial scenarios, tasks are often executed in an unpredictable event-driven manner rather than on a precise periodic schedule. Solely relying on periodic tasks for system performance assessment oversimplifies analysis and results in inaccurate conclusions. Therefore, industrial real-time embedded systems must be defined as event-driven and more research is needed to deal with event-driven applications.

In this context, low-overhead real-time operating systems (RTOSes) are commonly used for embedded industrial real-time applications. However, most of these RTOSes use static schedulers with preemptive priority-based schemes. This limitation hinders their optimal schedulability performance, particularly for event-driven industrial applications. The reluctance of RTOSes to implement dynamic scheduling can be attributed to concerns about high overhead. To address this issue, we propose a lightweight and low-overhead method to implement an event-driven dynamic scheduler. This method demonstrates that dynamic scheduling for event-driven tasksets can be implemented with lower overhead than typical static implementations. Moreover, our dynamic scheduler can be easily integrated into existing RTOS kernels. For instance, we demonstrate how to integrate our scheduler with FreeRTOS.

The key contributions of the paper are summarized below:

- Novel exploration of schedulers for event-driven embedded industrial applications by comparing static and dynamic policies.
- Provision of performance results for a real-world industrial application, including the impact in performance of AI tasks.
- Proposal of a lightweight dynamic scheduler that reduces both timing and memory overhead compared to its static counterpart.

The remainder of this paper is structured as follows. Section II offers a theoretical formulation and a motivational example

979-8-3503-2599-7/23/$31.00 ©2023 IEEE

for understanding scheduling processes. Section III discusses specific related work on real-time and event-driven scheduling, existing RTOSes, and dynamic scheduling implementation. The implementation of the dynamic scheduler and its integration into the RTOS kernel are detailed in Section IV. The experimental setup and results are presented and discussed in Section V. Finally, Section VI outlines the key conclusions of this work.

II. Preliminaries

This section presents a description and formulation of a system model to facilitate the understanding of the various parameters involved in the set of tasks of an application. In addition, a motivational example is provided to illustrate the timing discrepancies between static and dynamic scheduling.

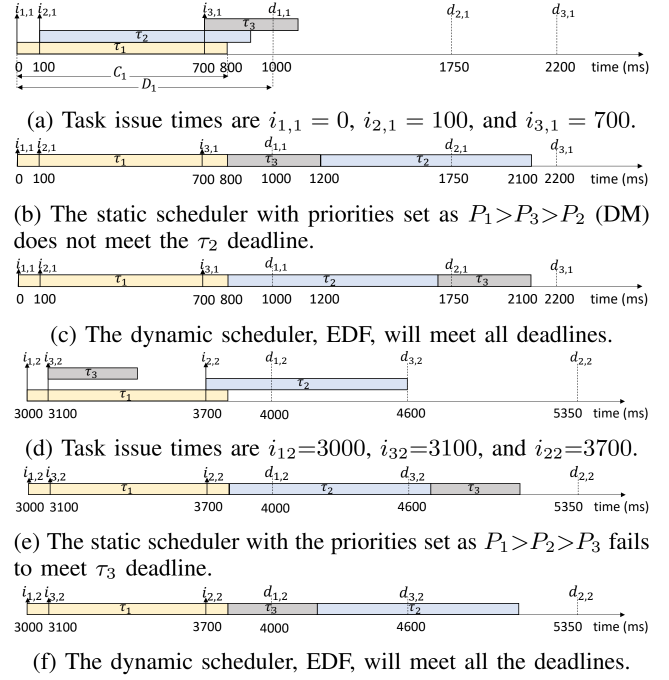
A. System model

In an event-based system, there is a taskset denoted by $\Gamma = \{\tau_1, \tau_2, ..., \tau_N\}$ that comprises N event-driven tasks. Each task is defined by two main parameters: $\tau_n = \{C_n, D_n\}$, where $n \in 1..N$. Here, C_n is the worst-case execution time of the task and D_n is its relative deadline, which refers to the time during which a task must be completed after its release. These τ_n tasks are exclusively activated and released through external or internal events. For instance, these events encompass external interrupts, soft timers, and interactions with other tasks. Each task's events arrive at specific issue times, denoted as $i_n = \{i_{n,1}, i_{n,2}, ..., i_{n,K}, ...\}$. Once a task is issued, the remaining worst-case execution time is represented by $c_{n,k}$. We define slack time as $s_{n,k} = D_n - c_{n,k}$, which is the amount of time that a task can be delayed without affecting the system's function. Each event $i_{n,k}$ is associated with an absolute deadline, calculated as $d_{n,k} = i_{n,k} + D_n$. Thus, if a task's completion time, denoted as $f_{n,k}$, exceeds $d_{n,k}$, it signifies a missed deadline for that particular event. With this formulation, a periodic task, τ_M, can be defined as a particular case of event-driven tasks where their events are issued with the same interval, denoted by $T_M = i_{M,j+1} - i_{M,j}$, $\forall j$.

B. Motivational example

In an event-based system, where tasks can be added to the ready-list arbitrarily based on the received events, it becomes crucial to use a dynamic scheduler that considers run-time parameters. Fig. 1 provides an illustrative example in which three tasks are defined with execution times and relative deadlines of $\tau_1 = \{800, 1000\}$, $\tau_2 = \{800, 1650\}$, and $\tau_3 = \{400, 1500\}$ (Fig. 1a). Their respective issue times are $i_{1,1} = 0$, $i_{2,1} = 100$, and $i_{3,1} = 700$, and their absolute deadlines are $d_{1,1} = 1000$, $d_{2,1} = 1750$, and $d_{3,1} = 2200$. Based on the relative deadline values D_n, task priorities should be set as $P_1 > P_3 > P_2$. The scheduling obtained with these priorities is shown in Fig. 1b, where the static scheduler fails to meet $d_{2,1}$. However, a scheduler with dynamic policies such as EDF or LST, which schedule tasks based on the closest deadline or the least slack time, meets all deadlines in this example(Fig. 1c).

In the case of using a different priority assignment ($P_1 > P_2 > P_3$) to meet previous deadlines, and different issue



(a) Task issue times are $i_{1,1} = 0$, $i_{2,1} = 100$, and $i_{3,1} = 700$.

(b) The static scheduler with priorities set as $P_1 > P_3 > P_2$ (DM) does not meet the τ_2 deadline.

(c) The dynamic scheduler, EDF, will meet all deadlines.

(d) Task issue times are i_{12}=3000, i_{32}=3100, and i_{22}=3700.

(e) The static scheduler with the priorities set as $P_1 > P_2 > P_3$ fails to meet τ_3 deadline.

(f) The dynamic scheduler, EDF, will meet all the deadlines.

Fig. 1: The set of tasks includes three tasks of $\tau_1 = \{800, 1000\}$, $\tau_2 = \{800, 1650\}$, and $\tau_3 = \{400, 1500\}$ for comparison between dynamic and static scheduling.

times ($i_{1,2} = 3000$, $i_{3,2} = 3100$ and $i_{2,2} = 3700$) as shown in Fig. 1d, the static scheduler does not meet $d_{3,2}$ (Fig. 1e). However, the dynamic scheduler is still capable of meeting all task deadlines (Fig. 1f). Therefore, irrespective of priority assignment or issue times distribution, the dynamic scheduler can meet all the timing requirements in this motivational example because it considers the time a task has waited to be executed after its event is received at run-time.

III. Related work

The literature on schedulability performance focuses mainly on periodic tasks [3]–[5], [7]–[9], neglecting the event-driven nature of tasks, particularly in industrial real-time applications. To include event-driven tasks in these periodic-based systems, they are often treated as periodic tasks. For instance, [9] considers event tasks as periodic tasks with a period equal to the minimum interval of their events. Similarly, in the deferable server and pooling methods [10], event-driven tasks are incorporated into the already scheduled system with periodic tasks. However, the literature lacks a detailed investigation of event-driven tasksets. The optimality of methods proposing task priority assignment is limited to special cases. For example, Rate Monotonic (RM) [3] and Deadline Monotonic (DM) [4] scheduling are not optimal if tasks are not issued together [9]. In [5], the authors proposed an algorithm that iteratively checks schedulability, which is still optimal in some scenarios, such as having an offset, where RM and DM fail. However, these methods are static and do not consider the dynamic nature of event-driven systems.

Numerous studies showcase the value of embracing the event-based nature for improved scheduling. Zhu et al. [11] propose event-driven scheduling for energy-efficient mobile

web applications. Yu et al. [12] introduce an event-driven sensor scheduling for unstable plants with wireless fading channels. Kong et al. [13] tackle event-driven scheduling for EV charging stations in the park-and-charge context. Villa et al. [14] address the issue of event-driven scheduling in small and medium-sized enterprises. These works demonstrate the potential benefits of incorporating event-based considerations in scheduling problems. However, these works do not target the scheduling of real-time embedded systems.

To effectively handle event-driven tasksets, the use of dynamic schedulers is crucial. Unix-based operating systems (e.g., LynxOS, TizenRT, OpenWrt, Apache NuttX) are adapted for embedded systems, offering varied scheduling policies, including SCHED_DEADLINE based on EDF in the Linux kernel [15]. However, these Unix-based schedulers might prove resource-intensive for small microcontrollers prevalent in embedded industrial scenarios. RTOSes, featuring leaner kernels and therefore less memory footprint, are designed to perform real-time operations with minimal overhead associated with managing tasks and system resources. However, a notable drawback of most RTOSes is the absence of dynamic scheduling support. For example, ERIKA Enterprise lacks full dynamic support in its accessible GitHub version. Zephyr employs two static parameters, namely priority and deadline, for task scheduling. In fact, most RTOSes use priority-based preemptive schedulers that support round-robin switching between tasks of equal priority (e.g., FreeRTOS, RT-Thread, MBed OS, MIPS Embedded OS (MEOS), Azure RTOS, DuinOS, SAFERTOS, VxWorks, and WinCE). Several of those use FIFO for tasks of the same priority (e.g., TI-RTOS, Deps, MQX RTOS, Huawei LiteOS, and Enea OSE). Notably, the priority ceiling, supported by some RTOSs to manage shared resources, is different from dynamic scheduling. Thus, there is a need to enable RTOSes with dynamic scheduling to provide optimal performance for event-based applications.

In this study, our aim is to integrate our proposed scheduler into the FreeRTOS kernel to achieve dynamic scheduling for event-driven tasksets. Previous research has explored ways to implement dynamic scheduling in FreeRTOS, such as the works by Kase et al. [16] and Paez et al. [17], which proposed a dynamic scheduling integration from an application perspective. Meanwhile, Belagali et al. [18] implemented a layer above the static scheduler that allowed changing the priority in run time based on dynamic parameters while keeping the scheduling parts unchanged. Although these are valuable contributions, the proposed methods resulted in a high-overhead implementation of dynamic scheduling in FreeRTOS. In contrast, recent research, such as the works by Salamun et al. [6] and Oliveira et al. [7], have improved previous approaches by implementing dynamic scheduling directly into the FreeRTOS kernel. However, their studies focused on randomly generated periodic tasks instead of event-driven tasks. Furthermore, their implementations resulted in higher overheads on the kernel, either in terms of timing by updating ready tasks in a time-based manner rather than an event-based manner [6], or in terms of memory by adding periodic-specific parameters and using multiple lists instead of one for ready tasks [7].

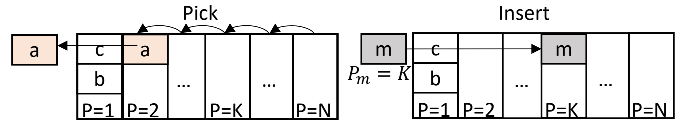


Fig. 2: Priority-based lists, a common implementation for managing ready tasks in static schedulers.

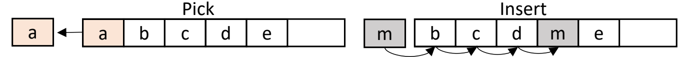


Fig. 3: A sorted list for minimal and low-overhead dynamic scheduling implementation.

IV. A Low Overhead Dynamic Scheduler

In this section, we describe the main characteristics of our low-overhead scheduler, namely: 1) use of a single sorted list of ready tasks; 2) event-driven updating of ready tasks for EDF-based scheduling. Furthermore, we offer insights into the seamless integration of this scheduler within industrial RTOSes, such as FreeRTOS.

A. Single sorted list of ready tasks

In a scheduler, a crucial aspect is managing ready tasks. Marked as ready to run, tasks join the ready list. During context switching, the scheduler selects the next task to run from this list. In static (priority-based) schedulers like FreeRTOS, handling ready tasks commonly involves utilizing multiple ready lists. Each list is assigned to a specific priority level, as depicted in Fig. 2. When a task is considered ready, such as upon receiving an event, it is inserted at the rear of its assigned list based on its priority, P_n, which can be accomplished with $\mathcal{O}(1)$ processing. Picking a task from these lists requires $\mathcal{O}(P)$ processing due to the need to check the lists one by one, in order of their priorities. The scheduler selects the next task to run from the first non-empty list encountered.

We propose a sorted list as a solution to manage ready tasks in a lightweight and low-overhead dynamic scheduler implementation. Priority-based lists are not suitable for dynamic schedulers, as tasks are prioritized at runtime. Moreover, multiple lists introduce more overhead, not only in memory but also in scheduling timing, compared to a single list. The resulting overhead is discussed in Section V-A. Fig. 3 illustrates the sorted list used for the selection and insertion of tasks. The list is always ordered, and the scheduler selects the top task in $\mathcal{O}(1)$ time. When a new task is inserted, the scheduler checks the parameters of other tasks in the sorted list and places the new task in its appropriate position based on the defined dynamic parameter. This operation requires $\mathcal{O}(N)$ processing for a linked list data structure, which is a preferred implementation considering the limited number of tasks in an embedded system. The dynamic parameter used for sorting is $d_{n,k}$ in the EDF implementation and $s_{n,k}$ in the LST implementation.

B. EDF scheduling with an event-driven update of ready tasks

Two methods can be considered for updating and reordering the ready list. The periodic update method involves checking, updating, and reordering the list at fixed intervals, whereas the event-driven method only updates the list when a new task is added to it. Salamun et al. [6] implement periodic updating of tasks' parameters in the ready list at every RTOS tick. However, this method introduces a large overhead in scheduling timing due to redundant processing. To fully leverage the event-based nature of tasks and reduce overhead, we propose updating tasks in an event-driven manner. Specifically, the dynamic parameter is updated only when a new task is inserted into the list and is sorted and placed in the list. However, periodic updates are necessary to implement LST. Therefore, we recommend using EDF to minimize overhead.

C. Integration in FreeRTOS kernel

After considering the aforementioned factors, we implemented a lightweight event-driven EDF-based scheduler in the FreeRTOS kernel. Our implementation is available in [19]. First, we added the parameters required for EDF to the Task Control Block (*tskTCB*). Each task is linked to a TCB, which includes attributes such as the task stack pointer. In contrast to [7], which added periodic-specific parameters like the period—thus not only making the scheduler exclusive to periodic tasks, but also creating additional memory overheads—we added only two key EDF parameters: $d_{n,k}$ and D_n. Then, we modified the functions to create tasks, such as *xTaskCreate* and *prvInitialiseNewTask*, to accept relative deadlines. Since we use a sorted list for ready tasks, the *pxReadyTasksLists* array used for priority-based lists was replaced by an ordered list. Subsequently, *prvAddTaskToReadyList* was modified to first update the inserted task's $d_{n,k}$ and then add it at the appropriate location in the list. Similarly, *taskSELECT_HIGHEST_PRIORITY_TASK*, which searched for a ready task from non-empty priority lists, was replaced with *taskSELECT_EARLIEST_DEADLINE_TASK*, which picks the first task from the ready list. In general, scheduling decisions are based on $d_{n,k}$ rather than P_n. For example, tasks that are blocked for a semaphore or a queue reception in *xEventList* are released after the event reception based on absolute deadlines rather than static priorities.

V. Experiments

This section presents the experiments and explorations conducted to evaluate the performance of dynamic scheduling utilizing the proposed scheduler. First, we assess the reduction in the number of $d_{n,k}$ misses in a general case scenario. The scheduler's efficiency in terms of timing and memory overhead is also analized and compared with its static counterpart. In the static scheduler, the P_n values are assigned based on DM. Furthermore, we explore a specific industrial use case based on Nespresso® coffee machines, focusing on the duty cycle of one of the most computationally intensive tasks in the application. We also evaluate the impact of different P_n settings in this experiment.

Of particular importance for industrial applications is achieving high processor utilization, which can reduce the overall expenditure on the platform. To address this, we conducted our experiments on the NUCLEO-G070RB board, a low-frequency, resource-constrained platform. Here, two scheduler versions, the proposed low-overhead EDF-FreeRTOS and native FreeRTOS, are implemented. The platform features an ARM Cortex-M0+ with $128\,\mathrm{KiB}$ of Flash memory and $36\,\mathrm{KiB}$ of SRAM. During the tests, approximately $32\,\mathrm{KiB}$ of SRAM are allocated for FreeRTOS heap memory. The processor frequency is set to $16\,\mathrm{MHz}$. The platform is equipped with several hardware timers that are used to measure the time during tests. STM32CubeIDE serves as the development toolchain.

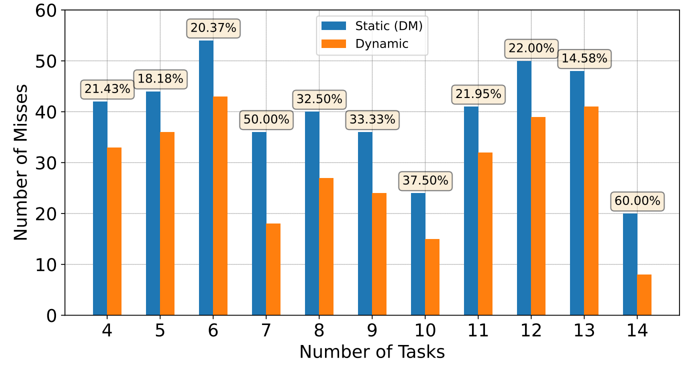


Fig. 4: Comparison between static and dynamic scheduling for event-driven applications of variable size Γ.

A. General scenario

In this scenario, all task parameters are generated randomly. The C_ns and S_ns are randomly assigned within intervals of $[200, 1100]\mathrm{ms}$ and $[500, 5000]\mathrm{ms}$, respectively. Based on the C_n and S_n generated for each task, its relative deadline is set as $D_n = C_n + S_n$. Furthermore, $i_{n,k}$s are randomly assigned to diversify test scenarios. Specifically, a maximum of 110 $i_{n,k}$s are generated within the interval $(0, 60000]$ ms. These settings are used to create test scenarios with varying task set sizes and requirements.

Fig. 4 presents a comparison of the schedulability performance between dynamic and static scheduling in terms of the number of missed $d_{n,k}$s. The observed improvement is quantified by calculating the performance gain using the following equation:

$$\frac{Misses_{static} - Misses_{dynamic}}{Misses_{static}}, \tag{1}$$

This metric is also shown in Fig. 4 on top of each bar representing different task sets. The performance gains range from $14.6\,\%$ to $60.0\,\%$. Furthermore, the performance gain is influenced by the parameters generated for each Γ. However, the dynamic scheduling approach outperforms the static scheduling approach in all event-based test cases.

Before analyzing timing overhead results, it's crucial to clarify various implementation-dependent factors. The principal source of scheduler timing overhead arises from context switching, encompassing three phases: 1) storing the current task's context, 2) choosing the next task to execute, and 3) restoring the context of the chosen task. The first and third

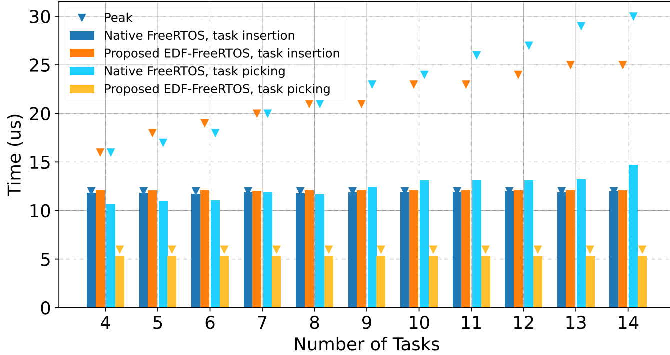


Fig. 5: Average time for the two primary scheduler overheads for both the native static scheduler in FreeRTOS and the proposed dynamic scheduler.

phases are identical in both schedulers. Thus, the second phase is the one that makes the difference when assessing timing overheads. Fig. 5 shows the timing overhead for task selection and picking, which is a critical aspect of the scheduler.

Another important factor in scheduling overhead is the time required to insert a task into the ready-to-use list. To compare the timing overheads between the static FreeRTOS and the proposed dynamic scheduler, the average time they spend on these operations is presented in Fig. 5. As Γ size grows, the task picking overhead in the static scheduler increases due to the $\mathcal{O}(P)$ cost described in Section IV. On the contrary, the proposed dynamic scheduler has a lower overhead, and increasing the task number does not have an effect due to the $\mathcal{O}(1)$ cost. However, the latter has slightly more overhead in task insertion than native static FreeRTOS. Although the proposed dynamic scheduler has a $\mathcal{O}(N)$ cost in task insertion, it is not significantly affected by the increasing size Γ and remains almost fixed. This is because only a small subset of tasks is in the ready-to-use list at any given time. To investigate the underlying cause, we present in Fig. 5 the peak time each scheduler incurred in task insertion and selection for each task set and scheduler. The results indicate that the maximum time that occurs during each test for the task insertion overhead increases as the size of Γ increases. However, given that the average time is almost constant, the instances where ordering and inserting a new task require more time than usual are infrequent. This situation typically arises when most of the system tasks are in the ready list, and the newly inserted task has a higher $d_{n,k}$ than most tasks, which requires more time to order and insert the task. On the other hand, priority-based lists commonly require traversing the ready lists down to lower-level priority lists, to ensure that a task with a lower P_n is always able to execute when it must run. As a result, the average task pick time increases nearly linearly as the task set size increases. On the basis of these findings, we propose that an ordered list is a better option, even for implementing static schedulers. Fig. 5 supports our argument.

The gain in timing overhead obtained can be calculated using the following equation:

$$\frac{(SP_{Native} + I_{Native}) - (SP_{EDF} + I_{EDF})}{(SP_{Native} + I_{Native})}, \qquad (2)$$

TABLE I: Typical tasks in a capsule coffee machine. We use artificial intelligence (AI) to generate long-running background workloads, keeping a maximum deadline of 100 ms to meet the reaction times typically accepted in this type of consumer applications [20]

Task	Type	Deadline	Period
Heater control	Event (Ext., Periodic)	6 ms	20 ms
Pressure pump	Event (Ext., Periodic)	6 ms	20 ms
Human-machine interface	Event (External)	80 ms	–
Flow control	Event (External)	95 ms	–
Brewing program	Event (Internal)	90 ms	–
Artificial intelligence (AI)	Event (External)	100 ms	–

where SP and I denote the average time spent selecting and picking, and inserting tasks, respectively. Based on the results obtained, this metric ranges from 22.7 % for $N = 4$ and up to 34.7 % for $N = 14$. Unlike other research works [7], our proposed scheduler implementation does not introduce timing overheads with respect to Γ size. This is a significant factor when considering the integration of the proposed dynamic scheduler for constrained industrial applications.

In terms of memory overhead, we measured the amount of memory required by the static scheduler to create a priority list. Specifically, 22 bytes are reserved for each list. Therefore, when $N = 14$ and DM is used to assign priorities in native FreeRTOS, this leads to an increase of up to 286 bytes compared to the proposed scheduler. In fact, the higher the priority levels required in native FreeRTOS, the more memory this part of the scheduler needs. For the dynamic scheduler, two variables per task are added for $d_{n,k}$ and D_n, resulting in an impact of up to 4 bytes per task. Thus, based on these scheduler-specific memory differences, the memory overhead saved in task and ready-list creation by using the proposed dynamic scheduler can be calculated as $\frac{900}{11} - \frac{100}{N}$ percent. Therefore, in our tests, it ranges from 56.8 % up to 74.7 % for $N = 4$ and $N = 14$, respectively.

B. Case Study: Coffee Machine

To evaluate dynamic scheduling's effectiveness in a real-world embedded industrial application, we examine the tasks involved in a Nespresso coffee machine designed for residential use. More recent models can also incorporate additional advanced features, frequently powered by artificial intelligence (AI), such as capsule recognition or voice control, reflecting a growing trend. These tasks, despite potentially having larger slack times, are characterized by significant execution times.

This section assesses and compares our dynamic scheduler with its static counterpart, while scheduling a mix of tasks that require fast response times, alongside long-running high duty-cycle tasks. The characteristics of the tasks considered for this example are presented in Table I. Tasks are event-driven, but they differ in terms of triggering events. The handling of human-machine interfaces, AI tasks, and flow control during brewing are triggered by external events at unpredictable times. The state machines controlling the brewing program respond to internal events. On the other hand, tasks such as the control of heating elements and pumps respond to external events issued periodically at 20 ms intervals.

Most of the tasks involved in a typical capsule-based coffee

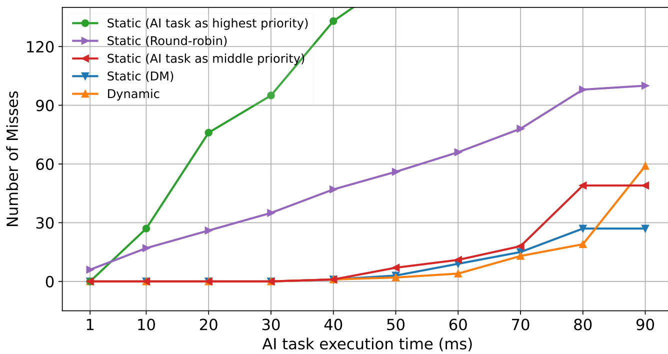


Fig. 6: Tasks deadlines misses in EDF-FreeRTOS and Native FreeRTOS with different task priority settings in a typical capsule coffee machine. The duty cycle for AI tasks varies from $1\,\mathrm{ms}$ to $90\,\mathrm{ms}$.

machine require only a few milliseconds to complete, and sometimes even just a few microseconds. However, the execution times for AI tasks are much longer due to their high computational load. To analyze the system's schedulability performance, we consider various workload scenarios created by background AI tasks with maximum reaction times of $100\,\mathrm{ms}$ [20]. Therefore, we sweep over a range of duty cycles for the AI task, varying from $1\,\mathrm{ms}$ to $90\,\mathrm{ms}$. A high-load scenario where consecutive external events are triggered at short intervals of up to $10\,\mathrm{s}$ is run on our experimental setup, which includes both static and dynamic schedulers.

Fig. 6 illustrates the number of deadline misses for each scheduler. Various priority assignment schemes are evaluated for the static scheduler, including deadline monotonic (DM), highest priority for the AI task, middle priority for the AI task, and round-robin with equal priority for all tasks. DM demonstrates the best performance among all static priority schedulers. However, the proposed dynamic scheduler exhibits superior performance in most cases. Notably, when the execution time of the AI task is fixed at $90\,\mathrm{ms}$ (with a maximum deadline of $100\,\mathrm{ms}$), static scheduling (DM and AI with middle priority) outperforms dynamic scheduling.

Fig. 6 also shows the pitfalls of a manual priority assignment. In particular, assigning the highest priority to the AI task produces the highest number of deadline misses, which shows a linear growth pattern. This occurs due to the frequent scheduling delays experienced by high-frequency, short-lived tasks such as heater and pressure pump control, which are triggered at intervals of approximately $\approx 20\,\mathrm{ms}$. Assigning similar priorities and using round-robin scheduling is ineffective as the scheduler lacks the ability to expropriate between them effectively, and the typical quantum time in FreeRTOS can be longer than the maximum deadline for some tasks.

VI. Conclusions

This work has presented an analysis of the performance of scheduling techniques for event-driven applications in the context of industrial applications. Our investigation covered both a general industrial applications scenario and then a very constrained real-world smart home device case study involving Nespresso coffee machines. Our findings indicate that dynamic scheduling is essential to achieve effective performance in these applications. Furthermore, our experiments revealed that compute-intensive tasks, such as DL inference, have a significant impact on scheduling performance. In future work, we plan to implement a DL-aware dynamic scheduler for optimal scheduling in these applications. Our proposal for a low-overhead event-driven dynamic scheduler to be integrated into RTOSes kernels has been demonstrated on the FreeRTOS kernel, with results indicating less overhead in timing and memory. We are confident that this paper provides the basis for further research into event-driven systems in the real-time community, which traditionally focuses on periodic tasks, to address the next generation of edge AI industrial systems.

Acknowledgment

This work was partly supported by a Ph.D. Student Grant for ESL-EPFL by Nespresso S.A. within the EPFL-Nestlé Collaboration Framework Agreement (EPFL TTO SoW no. 2019-0288).

References

[1] A. Akundi *et al.*, "State of industry 5.0: Analysis and identification of current research trends," *Applied System Innovation*, vol. 5, no. 1, 2022. [Online]. Available: https://www.mdpi.com/2571-5577/5/1/27

[2] G. C. Buttazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications*. Springer Science & Business Media, 2011, vol. 24.

[3] C. L. Liu *et al.*, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *JACM*, vol. 20, no. 1, pp. 46–61, 1973.

[4] J. Y.-T. Leung *et al.*, "On the complexity of fixed-priority scheduling of periodic, real-time tasks," *Performance evaluation*, vol. 2, no. 4, pp. 237–250, 1982.

[5] N. C. Audsley, "Optimal priority assignment and feasibility of static priority tasks with arbitrary start times," Department of Computer Science, University of York, Tech. Rep. YCS-164, 1991.

[6] K. Salamun *et al.*, "Dynamic priority assignment in FreeRTOS kernel for improving performance metrics," in *IEEE MIPRO*, 2021, pp. 880–885.

[7] G. Oliveira *et al.*, "Evaluation of scheduling algorithms for embedded FreeRTOS-based systems," in *IEEE SBESC*, 2020, pp. 1–8.

[8] V. P. Kumar *et al.*, "Dynamic scheduling algorithm for automotive safety critical systems," in *IEEE ICCMC*, 2020, pp. 815–820.

[9] R. I. Davis *et al.*, "A review of priority assignment in real-time systems," *Journal of systems architecture*, vol. 65, pp. 64–82, 2016.

[10] B. Sprunt *et al.*, "Scheduling sporadic and aperiodic events in a hard real-time system," Carnegie-Mellon Univ. Pittsburgh PA Software Engineering Inst., Tech. Rep., 1989.

[11] Y. Zhu *et al.*, "Event-based scheduling for energy-efficient QoS (eQoS) in mobile web applications," in *IEEE HPCA*, 2015, pp. 137–149.

[12] M. Yu *et al.*, "Event-driven sensor scheduling for mission-critical control applications," *IEEE TSP*, vol. 67, no. 6, pp. 1537–1549, 2019.

[13] F. Kong *et al.*, "On-line event-driven scheduling for electric vehicle charging via park-and-charge," in *IEEE RTSS*, 2016, pp. 69–78.

[14] A. Villa *et al.*, "Event-driven production scheduling in sme," *Production Planning & Control*, vol. 29, no. 4, pp. 271–279, 2018.

[15] D. Faggioli *et al.*, "An implementation of the earliest deadline first algorithm in linux," in *Proceedings of the 2009 ACM symposium on Applied Computing*, 2009, pp. 1984–1989.

[16] R. Kase, "Efficient scheduling library for FreeRTOS," 2016. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-204575

[17] F. E. Páez *et al.*, "FreeRTOS user mode scheduler for mixed critical systems," in *IEEE CASE*, 2015, pp. 37–42.

[18] R. Belagali *et al.*, "Implementation and validation of dynamic scheduler based on LST on FreeRTOS," in *IEEE ICEECCOT*, 2016, pp. 325–330.

[19] H. Taji, "ED-EDF FreeRTOS," https://doi.org/10.5281/zenodo.8256860, accessed 2023.

[20] B. W. Denkinger *et al.*, "Impact of memory voltage scaling on accuracy and resilience of deep learning based edge devices," *IEEE Design & Test*, vol. 37, no. 2, pp. 84–92, 2019.