Journal of
*Low Power Electronics and Applications*

MDPI

*Article*

# A Generalistic Approach to Machine-Learning-Supported Task Migration on Real-Time Systems †

**Octavio Delgadillo [1,*], Bernhard Blieninger [1,*], Juri Kuhn [1] and Uwe Baumgarten [2]**

1 Fortiss GmbH, Research Institute of the Free State of Bavaria, 80805 Munich, Germany; kuhn@fortiss.org
2 Department of Informatics, Technical University of Munich, 85748 Garching bei München, Germany; baumgaru@tum.de
* Correspondence: ruiz@fortiss.org (O.D.); blieninger@fortiss.org (B.B.)

† This paper is an extended version of our paper published in MCSoC 2021: Delgadillo, O.; Blieninger, B.; Kuhn, J.; Baumgarten, U. "An Architecture to Enable Machine-Learning-Based Task Migration for Multi-Core Real-Time Systems". In Proceedings of the 2021 IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC), Singapore, 20–23 December 2021; pp. 405–412. doi:10.1109/MCSoC51149.2021.00066.

**Abstract:** Consolidating tasks to a smaller number of electronic control units (ECUs) is an important strategy for optimizing costs and resources in the automotive industry. In our research, we aim to enable ECU consolidation by migrating tasks at runtime between different ECUs, which adds redundancy and fail-safety capabilities to the system. In this paper, we present a setup with a generalistic and modular architecture that allows for integrating and testing different ECU architectures and machine learning (ML) models. As part of a holistic testbed, we introduce a collection of reproducible tasks, as well as a toolchain that controls the dynamic migration of tasks depending on ECU status and load. The migration is aided by the machine learning predictions on the schedulability analysis of possible future task distributions. To demonstrate the capabilities of the setup, we show its integration with FreeRTOS-based ECUs and two ML models—a long short-term memory (LSTM) network and a spiking neural network—along with a collection of tasks to distribute among the ECUs. Our approach shows a promising potential for machine-learning-based schedulability analysis and enables a comparison between different ML models.

**Keywords:** task migration; real-time; ECU consolidation; RTOS; spiking neural network

**Citation:** Delgadillo, O.; Blieninger, B.; Kuhn, J.; Baumgarten, U. A Generalistic Approach to Machine-Learning-Supported Task Migration on Real-Time Systems. *J. Low Power Electron. Appl.* **2022**, *12*, 26. https://doi.org/10.3390/jlpea12020026

Academic Editor: Weidong Kuang

Received: 17 March 2022
Accepted: 28 April 2022
Published: 3 May 2022

**Publisher's Note:** MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

**Copyright:** © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

## 1. Introduction

The automotive industry is constantly evolving, especially with the aim to increase user comfort and achieve autonomy, but keeping safety as a priority, which is being enabled by the development of computer science and electronic technologies [1]. In fact, most of the innovations in the industry are related to those two areas [2]. As a result, computational requirements in modern cars are becoming greater. With the cost of electronics adding up to more than one-third of the total car cost [2], achieving an optimal usage of the computational resources becomes crucial for the automotive industry, where cost and resource efficiency is critical.

Until recently, the trend was to add many single-purpose devices for new tasks in the vehicle [3,4]. As a result, dozens of electronic control units (ECUs) can be found in modern cars. Nonetheless, if this number were to keep growing, this would soon turn into an unsustainable development, as it presents efficiency problems when considering the power consumption, cost, and weight of the car [1,4]. To counter this, ECU consolidation is pursued in the industry, with the aim of moving the execution of tasks to a few powerful, multiple-purpose devices [5]. This solution might seem obvious, and it is actually starting to be implemented in industry for some vehicle functionalities [6,7]. However, the global implementation of this strategy for all computational tasks executed in a vehicle faces

other challenges, especially when considering the diversity of the tasks, ranging from safety-critical ones to entertainment and comfort functions. Some of these challenges are maximizing the efficiency of the system, ensuring all tasks get to execute, mapping the tasks to the devices, and ensuring that all tasks meet their deadlines, especially ones that are highly critical.

At our research group, we explore a strategy for enabling ECU consolidation in the form of dynamic task migration. This means that tasks should be able to be executed on any device in a set and to be moved from one to another, depending on certain conditions, such as the load and functioning status of the devices, as well as the real-time properties of the tasks. This migration is split into two stages: the planning, responsible for mapping tasks to ECUs for executing them; and the execution, which actually performs the transfer of tasks from one device to another.

We consider that task migration planning can be enabled by using machine learning techniques for supporting the prediction on whether task sets distributed to each ECU are schedulable. This idea is motivated by the disadvantages of traditional methods, which can usually provide a fail-safe analysis of the feasibility of a schedule, but are often pessimistic and can lead to recurrent calculations [8]. Using such methods can become complex and computation heavy for larger dynamic task sets, especially if additional constraints are taken into account, such as the ones introduced by task dependencies and shared resources. We envision the usage of machine learning techniques to have the potential to provide fast and accurate predictions on the schedulability of task sets, even if at the expense of false positives. In fact, techniques explored at our research group have proven this approach to be promising, and other work has shown that machine learning can help speed up the schedulability analysis process in certain contexts [9]. Furthermore, using machine learning could enable predictions when uncertainties are present, as is the case for tasks with variable execution times or that interact with the real world, such as artificial intelligence algorithms and other automotive tasks.

The strategy we explore presents some challenges that require us to test the system characteristics extensively to evaluate its capabilities. First, in an automotive context, tasks with different levels of criticality and real-time constraints are present. Such constraints are very important, and they should be always met, especially for tasks that are safety critical. Second, introducing realistic uncertainty situations to the system could help showcase advantages in the machine learning approach over traditional schedulability analysis. Finally, it is necessary to compare the performance of different techniques to find the most suitable one for this application, as this is an area where evidence is missing. Therefore, to evaluate them, it is necessary to investigate the integration of a variety of machine learning algorithms and real-time platforms, especially under realistic circumstances. This work presents a setup for exploring a variety of machine learning techniques for predicting schedulability analysis as part of a system performing dynamic task migration between a set of ECUs.

The work in this paper is based on our previous work published in MCSoC 2021 [10] and extends it by describing the setup used to test the architecture introduced there, especially by describing the task deployment toolchain (Section 4) and the machine learning approaches used, which are the spiking neural network (SNN), briefly mentioned in the original publication, and a long-short-term-memory (LSTM) network, introduced here (Section 5).

The structure of the paper is the following. Section 2 presents related work concerning the idea of task migration, especially in the automotive environment, as well as other works exploring machine learning applications for scheduling. Section 3 presents an overview of the design and elements of the setup used for testing our developments, as well as a summary of the architecture presented in our original publication for MCSoC 2021. Section 4 introduces the central toolchain used for triggering the machine learning predictions and distributing the tasks to the ECUs. Section 5 shows the implementation details of the machine learning algorithms developed with this setup. The analysis of the

setup, the task deployment toolchain, and the machine learning models, along with the results obtained from its implementation and integration are presented in Section 6. The implications of these results are discussed in Section 7. Finally, conclusions are drawn and future work is mentioned in Section 8.

## 2. Related Work

The idea of task migration in a multiprocessor environment has been explored in research as a strategy for enhancing distributed systems. We build on this idea in our work for enabling ECU consolidation. Some works have explored task migration for the development of scheduling strategies that minimize task preemption and overhead, making it possible to execute a wider variety of tasks [11,12]. These examples, however, are mostly theoretical and compare their strategies to classic approaches, such as earliest deadline first (EDF). Further research has shown an implementation of task migration between ECUs with the creation of a framework for FlexRay systems, where a backup node is in hot standby to recover from failures [13]. This approach differs from ours because we focus on migrating individual tasks even while two nodes are already running, rather than migrating the whole task set at once from a device with a failure to an idle one.

Research based on machine learning has been performed to predict, improve, and secure WCET estimations, when complete WCET calculations are not derivable during design time, which is relevant to our research as our current machine learning models rely on such measured WCET values [14,15]. The usage of machine learning for solving the challenges of schedulability analysis has also been analyzed by researchers in several disciplines. Some early approaches investigated the usage of neural networks for analyzing schedulability during the design of the system, but the deployment of tasks in real-time was not in the scope of their work [16,17]. Other research on combining scheduling and machine learning has shown that it is possible to enhance scheduling algorithms such as EDF, especially in overload situations [18]. Furthermore, online machine learning approaches can improve the performance of energy-aware multi-core real-time embedded systems while saving energy [19]. Research on Ethernet TSN network scheduling showed that it can be enhanced to speed up the schedulability analysis by a factor of 5.7, while keeping a prediction accuracy of 99%, if machine learning evaluations are supported by traditional approaches when in doubt [9].

Recent developments have furthermore explored the usage of different machine learning techniques, such as reinforcement learning and shallow learning, to distribute tasks among several components [20,21]. However, more research is required on the idea of online task migration with machine learning. The most similar use case, to our knowledge, is a distributed system capable of offloading tasks to resources in the cloud, whereas the research in our group explores the usage in automotive real-time systems [22].

Finally, our publication in MCSoC 2021 [10] introduced the general architecture of the ECU system used in this setup. However, it did not provide a deeper understanding of the machine learning component and the deployment toolchain of the setup, nor did it compare the SNN approach to other approaches, such as the LSTM presented here.

## 3. Test Setup

In order to analyze our approach, we introduce an adaptable setup, which allows for testing improvements and alternatives in the different stages of the setup, as well as for testing the system under realistic and challenging situations. In this section, we first give an overview of the setup and a brief description of its components. Afterwards, we provide a brief insight on the implementation details of the ECU system used in our research.

### *3.1. Overview of the Setup*

The test setup for the system is shown in Figure 1. Its main elements are the following: a task deployment tool running on a central server, which generates task distributions, triggers the prediction on the schedulability of these distributions, and then deploys the best

one to the ECUs; a machine-learning-based schedulability analysis algorithm, responsible for predicting the feasibility of a task distribution; and the devices, which execute the tasks distributed by the deployment tool. Additionally, as part of the hybrid setup, a simulation of a vehicle is running on the server, allowing for testing and illustrating the system's interaction with an external environment.

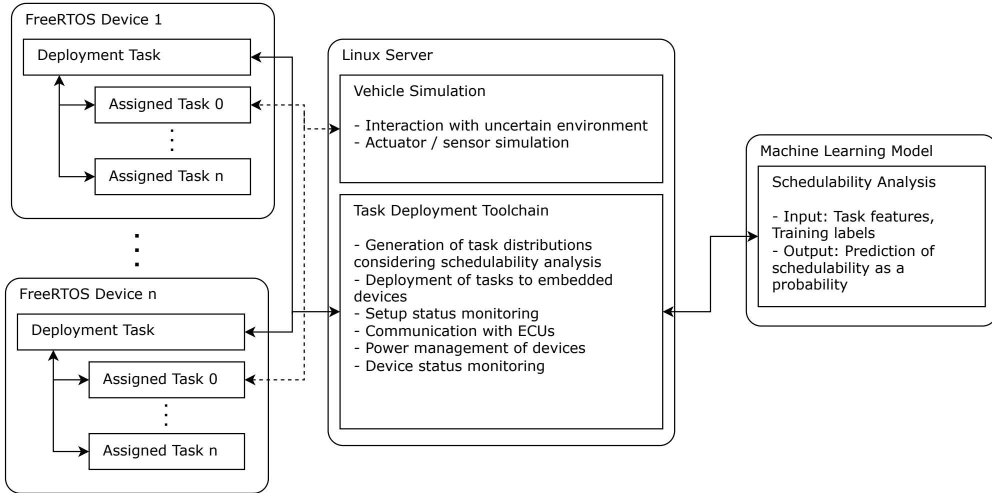


**Figure 1.** Test setup overview. Arrows depict the interaction between different elements.

The task deployment tool is the center of the setup. It is responsible for communicating with the devices, controlling the execution of the tasks, and checking the device status for reacting to hardware or schedule failures. This tool is also responsible for generating new task distributions according to the status of the devices, triggering prediction using the machine learning component for evaluating the best distribution, and performing online training on the machine learning using runtime data. The deployment tool is built in a modular fashion, so it is also adaptable to work with different schedulability analysis algorithms and ECU implementations.

The devices interact with the deployment tool for receiving a set of tasks to be executed and for sending back the status of the running tasks, as well as the general device status. Furthermore, the devices can communicate separately with the simulated vehicle in case the execution of a task needs interaction with elements outside the ECU, such as sensors and actuators. The architecture of these devices was introduced in our original publication and is therefore only covered briefly in this work.

The setup makes it possible to test different machine learning algorithms in the prediction phase, and in this work, we present two different strategies: a spiking neural network (SNN) running on a SpiNNaker neuromorphic hardware and a LSTM network running on a Nvidia Jetson TX2 board (Section 5). This means that the machine learning runs on a dedicated hardware, while the ECUs only execute the distributed tasks. Even if at this point we have only collected evidence with these two models, this setup allows for comparing the performances of a wide variety of machine learning approaches for distributing tasks to ECUs during runtime.

The simulation used in the current setup is very simple, but also extendable. It consists of a wheeled robot exploring a labyrinth, which integrates a LiDAR to sense its surroundings and communicates with the device running the respective control task over the network. To achieve this, sensor data are shared and then processed to produce a control signal, meant to make the robot autonomously explore the labyrinth. This simulation is hosted on a Linux server and allows for testing the other elements of the setup in an environment that introduces challenges similar to the ones found in an actual vehicle.

### 3.2. ECU System Implementation

The ECUs are responsible for executing the real-time tasks assigned to them by the task deployment toolchain. The details on their architecture and the tasks executed were presented in the original publication [10]. Therefore, here, we only provide a summary as a context for the rest of the work presented.

The embedded devices used as ECUs in our setup are ARM-based development boards with a Xilinx UltraScale+ MPSoC. This MPSoC contains two processors: an ARM Cortex-A53 with 4 cores and an ARM Cortex-R5 with 2 cores, of which only core 0 of the A53 was used for this work, but an extension to all available cores is planned for future work. The real-time operating system used was FreeRTOS. This decision was motivated by its popularity and large user community, the presence in the industry, its tiny kernel, and its wide range of applications, ranging from the automotive to the IoT domains. Its features, such as the possibility of implementing different scheduling algorithms and time measurement, allow us to easily test the capabilities of the rest of the setup.

To test the distribution capabilities of the setup, a set of suitable tasks was developed for the platform. They are described in Section 3.2.1. The scheduling algorithm used for the execution of such tasks is described in Section 3.2.2. Finally, a brief explanation on how the boards receive and monitor the distributed tasks is provided in Section 3.2.3.

#### 3.2.1. Tasks

Two types of tasks are present in the system: simulated tasks, meant to test the system under realistic conditions, interacting with an environment, and to show in the simulation when they miss their deadlines; and automatically generated dummy tasks, which allow for filling the utilization of the ECUs and testing the setup with a range of different worst-case execution times and periods. In this work, we only present one simulated task, which interacts with the simulation described in Section 3.1. All tasks in this work are periodic with relative deadlines equal to their respective periods, and they may execute for a specific number of jobs or continue their execution indefinitely unless the distributed task set changes. They can also be started and stopped at any time, as requested by the central deployment tool.

The implemented simulated task receives sensor data from the robot in the simulation and provides a control signal to autonomously navigate the labyrinth. To achieve this, the task implements a SLAM algorithm and calculates a route to the nearest non-visited spot. This task runs on an ECU and communicates with the simulation over Ethernet, exchanging sensor data and control signals. The simulated task helps to find shortcomings in the migration strategy, such as the current lack of a stateful migration process, causing the navigation progress to be lost if it moves to a different ECU.

It is worth noting that the execution time of this task has a considerable variability. Depending on communication and environment uncertainties, the worst-case execution time for one job was found to be around 550 ms, while the average execution time was only around 210 ms, both calculated using software end-to-end measurement. We assume this to be beneficial for our research, as we expected the machine learning algorithm to be able to learn to cope with tasks with variation in their execution.

The dummy tasks are created automatically, using an extension of the task set generator from the COBRA framework [23]. The tasks produced have pseudo-random periods and execution times. The actual execution times are based on a combination of benchmark programs from the TACLeBench suite [24]. The tasks produced with our extension of the COBRA framework cover a range of 100 ms to 10 s, making it possible to test the system with various task utilization values and a total utilization approaching 100% on all devices. Such high utilization rates of the devices are provoked on purpose, in order to generate training data labeled as not schedulable, as well as to cause unstable system states that trigger the automated and dynamic task migration. Additionally, the generated tasks introduce some level of dependency as TACLeBench programs use shared resources. This introduces an additional challenge for the machine learning models.

The execution times and periods of the dummy tasks, in contrast to the ones of the simulated task, are consistent and relatively precise. However, our extension to COBRA uses an estimation for the execution time of the benchmark programs on the ARM64 platform, performed with averaged software end-to-end measurements, while the original framework works with a formal calculation of CPU cycles for every task, but only provides information for the x86 architecture. The measurement strategy used might be inaccurate, but we assumed the usage of machine learning would allow overcoming the disadvantages of an estimation, while avoiding a time-expensive exact worst-case execution time analysis.

#### 3.2.2. Scheduler

The machine learning component enables the adaption of the setup to any scheduling policy. However, our research focuses on real-time systems, and for this reason, one of the most commonly used real-time scheduling policies, earliest deadline first (EDF), was selected as the policy for scheduling the execution of tasks in each of the ECUs, considering that at the moment, only a single-core ECU implementation is used. The main motivation for picking EDF over other common real-time scheduling policies, such as the static rate monotonic (RM) policy, is that it is optimal for single-core systems when combined with task preemption, meaning that if there exists a feasible schedule for the analyzed task set, EDF will always find one [8,25]. Moreover, in an ideal situation, where no overhead for the dynamic calculation and the context are considered, all task sets with an utilization smaller than or equal to 1 are feasible under this policy. Furthermore, even if they are taken into consideration, a preemptive EDF scheduler adds up less implementation overhead than RM, due to enforcing less context switches, which usually adds a larger overhead than the dynamic priority assignation [26].

In the ideal case, there would be no need for a machine-learning-aided schedulability analysis, as adding the utilization of the tasks in a task set would provide a quick analysis. However, the actual implementation adds an overhead due to task dependencies, context switches, and priority calculation. Thus, we assumed that a rich data set for training the machine learning model would allow it to cope with uncertainties such as this overhead and the variable execution times of the tasks. Our implementation includes therefore an EDF scheduler extended for reporting and counting when tasks miss their deadlines or finish their jobs correctly, as these data are used by the deployment toolchain to predict or to train the model. Furthermore, the scheduler used makes it possible to add and delete tasks from an ECU's running task set on runtime and to execute both limited and unlimited numbers of jobs for every task.

All of the deployed tasks are executed with the EDF scheduler, but higher-priority tasks, such as the monitoring ones, are not considered by it and are therefore scheduled by the FreeRTOS base scheduler.

#### 3.2.3. Task Monitoring

The monitoring process for the tasks involves receiving the distribution of tasks sent by the deployment toolchain, preparing the tasks for their execution with the EDF scheduler, checking the status of the running tasks, and finally, reporting it back. It consists of two sub-tasks, one only responsible for communicating with the deployment server and the other for the rest of the process. Both sub-tasks are simple in order to reserve processing power for the execution of the deployed tasks.

The process works as follows. Communication with the deployment tool occurs via the exchange of JSON messages containing relevant information, either for receiving a new distribution or for sending the status of the running tasks. The tool tells the ECU to start or stop tasks and sends their execution details (such as their ID and real-time properties). Once received, tasks are created and queued for their execution with the EDF scheduler and can also be terminated if requested by the tool. During runtime, the status of every running task is periodically checked, storing data such as the number of completed jobs

and deadline misses. When the deployment tool requests the ECU to report its status, the information collected from the tasks, along with the general ECU status, is sent back.

## 4. Task Deployment Toolchain

The task deployment toolchain is the central element in the setup. It was originally designed and implemented by Bernhard Blieninger and Robert Hamsch in the scope of a student's thesis. The toolchain is responsible for distributing the tasks to the ECUs according to the system load and the status of each ECU, also referred to as device or board. To achieve this, it generates several different task–board distributions, also referred to as task–board configurations (TBC), and selects them for deployment. Within this work, the toolchain was extended to work with the ECU architecture based on FreeRTOS and the spiking neural network to fully show its potential and general applicability.

The toolchain was designed with three main modules: one central tool and two interface layers, one for the machine learning model (ML layer) and one for the ECU operating systems (OS layer), which interact with each other as shown in Figure 2. This modular design is intended to be able to support a wide variety of different platforms and ML implementations. The core tool is responsible for instantiating the OS and ML layers, with their ECU and ML-specific components. It also triggers the generation of task–board distributions and evaluates the system status and machine learning predictions, in order to select and deploy the best-suited one. The OS layer is responsible for all the interaction with the ECUs/boards, including power management of the boards via a PoE-capable switch, the deployment of tasks to execute, and the status monitoring of applied tasks and available devices. Finally, the ML layer is responsible for loading the machine learning model, performing the predictions for each task distribution (on external hardware), and reporting the result to the central tool. Each of the subcomponents is described in further detail in this section.

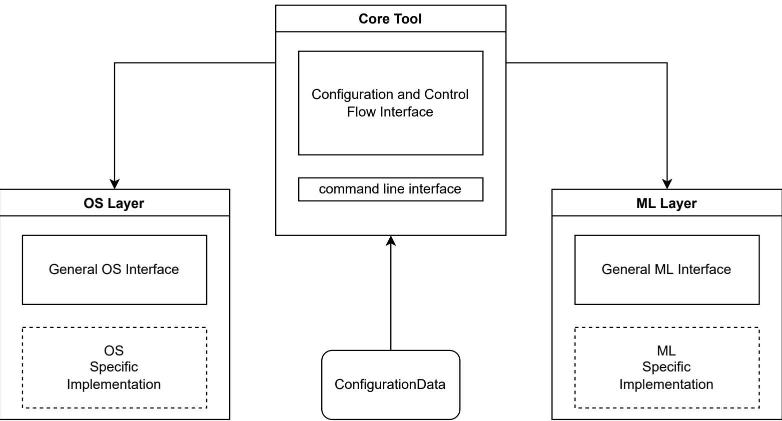


**Figure 2.** Diagram depicting the components of the toolchain.

While the specific behavior of each of the components is described in the next subsections, the general behavior of the toolchain is shown in Algorithms 1–3 and briefly explained next. First, as shown in Algorithm 1, no active task–board configuration exists upon the start of the tool, so a task–board configuration is initialized with data from the configuration file. Then, according to the configured strategy, an initial distribution is generated and set as the active configuration. The possible strategies are Even (each board gets distributed an equal amount of tasks if possible) and Fewer Boards (the smallest amount of boards possible is used). The execution continues with Algorithms 2 and 3. In Algorithm 2, first, the reported runtime data are checked for unstable (that is, tasks risking missing or already missing deadlines) or broken status, which would mean that the active configuration has an issue. If that is the case, the TBC with the next-best score is chosen as the active configuration and deployed. Then, if new runtime data from the boards are available, online training is triggered for the machine learning model. If no alternate TBCs

exist, they are prepared in advance and added to the list of TBCs for their prediction. This list is formed by generating TBCs that differ from the active TBC distribution by a task delta $\delta$ for each board, where $\delta$ is the task difference for a board, and $k \leq \delta \leq n$. This allows for optimizations in the TBC generation, as well as the task migration. Algorithm 3 describes how the list of possible TBCs is evaluated for selecting the next active configuration. For every TBC, the task distribution for each board is handed to the machine learning model as the input for a new prediction. A global score is generated for each TBC by merging the predictions from different boards and adding a bonus according to the strategy (Even or Fewer Boards), a process that is described in more detail in the next paragraph. Finally, the TBC with the best score is assigned as the new active configuration, and the OS layer deploys the tasks to the boards.

**Algorithm 1** Simplified algorithm describing the initial start and strategies of the task deployment tool

```
    if no_active_configuration then
2:      tbc = read(configurationFile)
                                        ▷ Strategy: assign all tasks to first board
4:      if strategy is FewerBoards then
            for t in tasks do
6:              add_task_to_board(tbc.boards[0], t)
            end for
8:                                 ▷ Strategy: distribute tasks among all active boards
        else if strategy is Even then
10:         i = 0
            for t in tasks do
12:             add_task_to_board(tbc.boards[i%size(tbc.boards)], t)
                i = i + 1
14:         end for
        end if
16:     tbc_list.append(tbc)
        active_configuration = tbc
18: end if
    — Continue with Algorithm 2
```

**Algorithm 2** Simplified algorithm for board status check, online training, and configuration changes

```
    for b in tbc.boards do
2:      if b.status is unstable or broken then
            active_configuration = tbc_list.best_tbc
4:          deploy(active_configuration)
            break
6:      end if
    end for
8:  if active_configuration.collected_runtime_info then
        train_ml_model(active_configuration.collected_runtime_info)
10: end if
                      ▷ Generate alternate configurations by moving a few tasks around
12: if no_alternate_tbcs then
        new_tbcs = generate_tbcs_with_task_difference(active_configuration, k, n)
14:     tbc_list.append(new_tbcs)
    end if
16: — Continue Algorithm 3
```

**Algorithm 3** Simplified algorithm for prediction, scoring, and deployment of TBCs

```
                                        ▷ Trigger prediction for each board in TBC
 2: for tbc in tbc_list do
        for b in tbc.boards do:
 4:         b.prediction = predict(b)
        end for
 6:     tbc.score = evaluate_predictions(tbc.boards)
                                        ▷ Bonus according to strategy
 8:     if strategy is FewerBoards then
            bonus = apply_bonus_fewer_boards(tbc)
10:     end if
        if strategy is FewerBoards then
12:         bonus = apply_bonus_even(tbc)
        end if
14:     tbc.score = tbc.score + bonus
    end for
16: sort(tbc_list)
    active_configuration = tbc_list.best_tbc
18:                 ▷ Deploy tasks to board. Boards report status in the background
    deploy(active_configuration)
```

The modularity of the toolchain enables the implementation of different scoring mechanisms, concerning the weight of the predictions, as well as the scoring based on a global scheduling strategy. Thus, it is possible to optimize the score according to the strategy mentioned before, giving a respective bonus to distributions that approach an equal task-to-board distribution between all running boards (Even) or that maximize board utilization and minimize the number of involved boards (Fewer Boards). Further optimizations are possible by merging the predictions for every board into a single score, either by averaging them to obtain the best overall score or by punishing bad predictions for each task–board configuration in a distribution set. The prediction and scoring algorithms run in the background, so that whenever new runtime-generated information on the active configuration is available, predictions are updated for the best alternative task–board configurations. This allows the system to redistribute tasks among the available ECUs in the case of failure or unstable system behavior, for example from hardware failures or missed deadlines. Furthermore, manual reconfiguration of the available boards and the addition of new tasks can also trigger immediate predictions and the application of new task distributions to the boards.

### *4.1. Core Tool*

The core tool controls the flow described in the algorithms mentioned before. It also acts as the interface module for the whole toolchain, combining the OS layer module with the ML layer module, making it possible to evaluate ML predictions, as well as to execute the task deployment. In addition to the functionality described above, the core tool is also capable of reading a starting configuration with OS, hardware, and ML-specific parameters, which can be changed during runtime, triggering a manual reconfiguration. This feature, along with some external control signals, allows for controlling the toolchain with external code or via the command line interface, thus integrating it in a larger setup, where another element may request the execution of new tasks or change the configuration of available ECUs. For the purpose of testing this feature in this work, configuration changes were simulated with predefined stories, where tasks and boards are added or removed at runtime, forcing the whole system to react in real-time and redistribute the tasks.

### 4.2. Operating System Layer

The OS layer implements core functionality for generating the task–board configurations and deploying them on the boards. Such configurations are kept in a list and are accessible from the core tool to run the machine-learning-based schedulability analysis. It is also responsible for implementing the communication with the boards and their power management (PoE-based hard reset), as well as storing the information on the status of each board and the tasks it is running. This information is then available to the tool for detecting unsuccessful distributions and online training of the ML model.

This layer includes a few interchangeable elements that allow for integrating different hardware and operating systems in the setup: one implementing communication features, one defining a device with such an OS, one defining information for tasks in that OS, and one defining information for a single task set on a device. The communication element is the one implementing all the interaction with the ECUs.

### 4.3. Machine Learning Layer

The ML layer implements the functionality for evaluating the task–board configurations generated previously in the OS layer. Therefore, a first step is loading the machine learning model from its saved state (preferably after a pre-training or from a previous run). Then, the input data are prepared and converted into the respective input features for the active ML model. With the input features ready, either the prediction or the training process is executed, depending on the request sent by the tool. Finally, the output data are converted to a score prediction for each task–board distribution analyzed, and the scores for all boards are aggregated. This layer is also capable of storing the updated state of the active ML algorithm upon shut-down of the system, enabling the persistence of learning.

This interface layer has to be adapted to the selected machine learning algorithm and is interchangeable upon the start of the tool. Currently, the setup has been tested extensively with the ML models shown in this work. While it is possible to combine different OS and ML layers per design of the toolchain, a combination of ML models not trained for the ECU component will tend to be unreliable. Therefore, the selection of the machine learning algorithm in the toolchain is not independent of the selection of the ECU operating system, as the training and prediction circumstances must be compatible.

## 5. Machine Learning

The machine learning algorithm is responsible for performing a prediction on the schedulability of a set of tasks on a single ECU. The result of this stage is then evaluated in the tool with predictions for other ECUs. As mentioned before, the architecture of the setup is designed for flexible use of machine learning components. As part of the work presented here, we have trained and tested a long-short-term-memory (LSTM) network and a spiking neural network (SNN) with the runtime data produced by the FreeRTOS-based ECUs.

Both of the implemented networks run on dedicated hardware, allowing for a better performance of both the machine learning algorithms and the rest of the setup. The LSTM model runs on a Nvidia Jetson TX2 board, while the SNN runs on a SpiNNaker SpiNN-5 [27]. Both of these platforms were chosen to support lightweight embedded use.

The exploration of a neuromorphic architecture over other traditional shallow or deep learning techniques was motivated by the potential advantages of neuromorphic computing over traditional ones, in particular in machine learning applications. These advantages include lower power consumption, scalability, parallelism, and better performance in some applications [28,29]. In particular, the possibility of achieving lower power consumption is attractive for an embedded application.

### 5.1. LSTM Network

In order to compare and evaluate the results of our previous work [10], we designed an LSTM network that takes the task parameters for predicting the feasibility of a task set. The features extracted from the task parameters and fed to the LSTM model as input are

the priority of the task, its utilization, its relative deadline, and the number of jobs to be executed for a predefined maximum of eight tasks per task set. It is worth noting that in the setup described here, the priority value of a task is irrelevant, as the EDF scheduler assigns priorities dynamically; however, it could be used in the future for coupling to other scheduling policies with fixed priorities or for mixed criticality scenarios. The task features are first normalized, and then, in order to mimic a task scheduling scenario, the network is fed with them in a task-by-task manner in eight time steps.

The network architecture is pictured in Figure 3. It is designed with 32 input nodes followed by two LSTM layers with 64 and 128 nodes, ending with three dense layers with 128, 256, and 512 nodes, respectively, and a ReLU activation function. Each layer is also equipped with a drop layer (0.3), leading to a final dense layer with a sigmoid activation function. For the training, a binary cross-entropy loss function was used, along with the Adam optimization algorithm, setting the learning rate to $10^{-5}$ and the decay to $10^{-7}$.

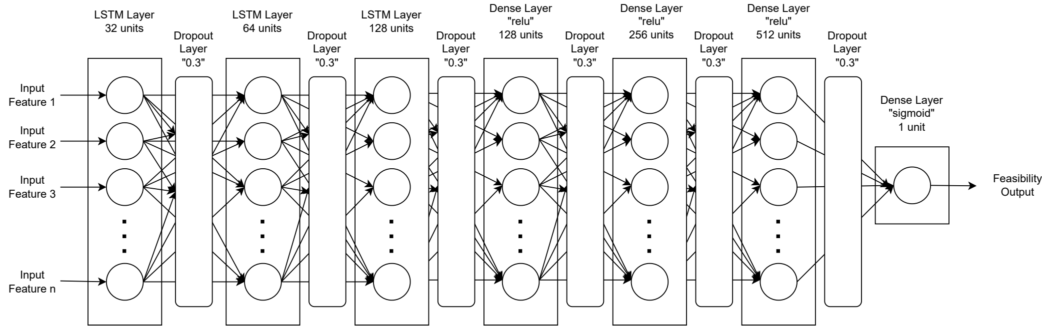


**Figure 3.** Architecture of the LSTM network.

### 5.2. Spiking Neural Network

For designing the spiking neural network, the first step was selecting the neuron and synapse type. The neurons were chosen to be leaky-integrate-and-fire (LIF). This type of neurons get charged over a span of time with input spikes, and once they reach a threshold, they send an output signal, while they get discharged when no spikes occur; this behavior can be described by an exponential function [30]. The inputs can be excitatory or inhibitory, meaning that they can increase or reduce the excitation level of a neuron. For the synapses, spike-time-dependent plasticity (STDP) was chosen, as this allows for unsupervised and semi-supervised learning. In this kind of synapse, the weight is adapted depending on the time difference between spikes at the input and the output of a neuron, increasing the weight for inputs that receive spikes immediately before an output spike is generated and reducing it for inputs that have spikes immediately after or a long time before an output spike.

Afterwards, the same features as for the LSTM were selected, that is the priority value, the utilization, the relative deadline, and the number of jobs for each task running on the ECU, up to a maximum of eight. To input these numerical values to the SNN as spike trains, so-called rate coding was used, where a higher numerical value is represented as a higher rate in the spike train. For converting the features to rates, they were first normalized and then converted to a rate in the range of 0–40 Hz. The network was chosen to be a probabilistic classifier where the average rate of the output spikes represented the probability of the task set failing.

The network implemented is shown in Figure 4. It is relatively simple, especially when compared to traditional multiple-layer neural networks, as it only consists of an input population with 32 neurons and an output population with 20 neurons, connected in an all-to-all fashion with excitatory STDP synapses. To train the network, a population of 20 teaching neurons with static inhibitory synapses (that is, with fixed weights) was used. These neurons were connected in a one-to-one fashion to the output neurons, inhibiting output spikes when the sample was labeled as successful. After each training sample, the

weights in the synapses were adapted automatically by the framework according to the STDP rule.

**Figure 4.** Architecture of the spiking neural network.

## 6. Results and Analysis

The system described in this paper and the corresponding results can be analyzed best by first performing a separate analysis on each of the elements in the setup and then a global analysis on their integration as a single element (this is, the test setup). Hence, this section is divided into three subsections, first analyzing the test setup, then the deployment toolchain, and finally, the machine learning approaches explored, along with their results.

### *6.1. Test Setup*

The analysis of the test setup was performed by integrating and testing the different elements. The simulation allowed graphically demonstrating the impacts of a failed ECU, for example due to a faulty behavior in the implementation of our system adaptions and tasks or due to deployed task sets not being schedulable. This makes it easier to illustrate, evaluate, and compare different ML approaches and different ECU system implementations, as well as the improvements performed on them.

### *6.2. Task Deployment Toolchain*

To analyze the performance of the toolchain, it was extended to work with the mentioned ECU system and the two ML approaches. This allowed us to further exploit and test different aspects of the toolchain as described next.

Extending the toolchain to integrate the FreeRTOS system allowed it to show not only its adaptability to different ECU implementations, but also its functionality. First, it is capable of generating different task–board distributions and deploying the most promising one to the ECUs, as well as reacting to changes in the available boards and the tasks to schedule. Second, it is capable of reacting to failures in the available boards and to deadline misses in the deployed configuration. Third, it is able to reduce the number of task migrations when changing the active task–board distribution by first exploring

configurations with a small task difference. Finally, it allows for implementing ECU system-specific functionality, such as predicting failures when job-specific data are provided by the ECU.

Testing the integration of the SNN and LSTM models with the toolchain further showed its adaptability to different machine learning components. Additionally, it was also possible to demonstrate that the predictions can be performed at runtime, while almost neglecting the runtime for a prediction as the toolchain uses pre-evaluation by design. Furthermore, it was possible to perform online learning on both models using the collected data from the toolchain to improve the model. Finally, the logging capabilities of the toolchain proved to be useful for analyzing the performance and for debugging the machine learning model with the running ECUs.

### 6.3. Machine Learning

Both the LSTM and SNN models were integrated with the system, performing live predictions on the schedulability of the deployments. Their performances were evaluated in two regards: first, the accuracy of the predictions; and second, the time necessary for performing them.

First, it is relevant to mention how the data set used for training and testing the models was generated, as this is the basis for the analysis of the two models. These data were generated by deploying random sets of tasks to the ECUs, letting them run for a time span of 60 to 120 s, and then, collecting the data reported by the ECU system on the deadlines missed and the number of finished jobs. As each task set represents one sample, the generation of samples is slow, and therefore, at the time of developing the networks, the task set was relatively small (6000 samples), as mentioned in the original publication [10]. Further data generation allowed us to collect a total of around 29,900 samples. This amount might still be small for achieving optimal results in the networks, but allows for testing the approaches and showing their potential. The analysis for each of the models is presented separately next.

#### 6.3.1. LSTM Network

The model was trained, validated, and tested in the target hardware. First, the original data set was used, divided into 4500 training and 1500 validation samples. This produced a training loss of 0.373 and an accuracy of 0.807. The validation resulted in a loss of 0.277 and an accuracy of 0.895. For the training, early stopping of epochs was configured, stopping around epoch 79, taking around 3 s per epoch, and around 615 µs/sample. Results from the test with an additional 1800 samples revealed a loss of 0.236 and an accuracy of 0.877 with a prediction time of 249 µs/sample. Figure 5a shows the confusion matrix for this run.

Afterwards, a run of the same network on the larger data set was performed, using 25,700 training samples and 3000 validation samples. This second evaluation resulted in an early stop after 14 epochs for around 13 s per epoch (517 µs/sample), with a training loss of 0.255 and a training accuracy of 0.884. Validation loss in this case was 0.217, while the validation accuracy achieved was 0.899. The test was performed with 1200 test data, revealing a prediction accuracy of 0.883 and a loss of 0.294 taking 236 µs/sample.

#### 6.3.2. Spiking Neural Network

After training the net with the initial data set, using 4500 samples and using a test set of 1500 samples, the test prediction accuracy of the net was 0.850. The confusion matrix showing the performance of the SNN is depicted in Figure 5b. Regarding the speed, it was limited by two factors. First, the smallest time step possible with the SpiNNaker is 1 ms, meaning that in our current implementation, every sample runs for 200 ms. Second, the network designed in a traditional architecture has to be converted into a network in the neuromorphic architecture, and then, the results are converted back. This is performed automatically by the SpiNNaker board, but took a minimum of around 40 s for every run for the current architecture, which is particularly large when running with few samples.

In the case of 6000 samples separated into training and test, the total runtime was around 22 min and 18 s, averaging around 233 ms/sample.

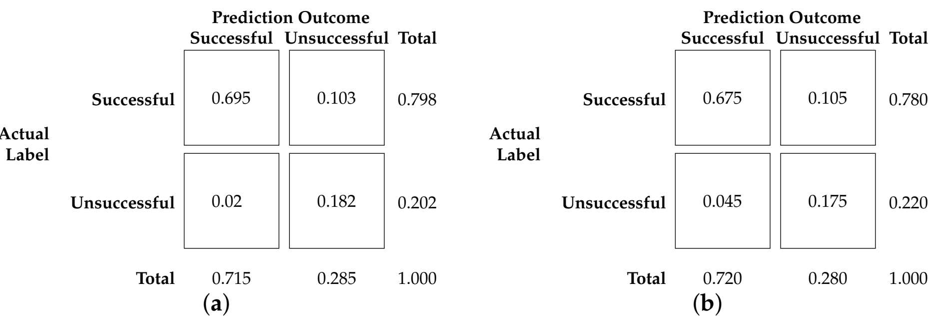


**Figure 5.** Confusion matrices showing the performance of the LSTM and SNN models on the test data. (**a**) LSTM network. (**b**) Spiking Neural Network.

However, the performance obtained with the larger task set, divided into around 24,000 training samples and 5900 test samples, was worse than with the initial one. Multiple runs with these data provided an accuracy between 0.793 and 0.815, which is a considerable drop. This behavior is contrary to the normal machine learning expectation, where a higher amount of data normally leads to an increase of prediction accuracy. As the accuracy of the LSTM also did not drastically improve, we assumed that the data set needed to be further extended. In addition, we discuss alternative improvements in Section 7. The time spent for this run was 1 h, 49 min, and 20 s, averaging 219 ms/sample.

Because the application is strongly bound to safety-critical constraints, it is worth mentioning that, in addition to the accuracy, it is desired to keep false positives (false successful predictions) as low as possible. False positives in this case will result in unfeasible task sets, causing the system to fail. In the case of the models presented here, there is also room for improvement, as shown in the confusion matrices in Figure 5.

## 7. Discussion

The approach described in this paper provides the necessary tools for evaluating the performance of a variety of machine learning techniques for predicting schedulability analysis and enabling dynamic task migration. The setup also enables an easier testing and integration of improvements and extensions to the ECU architecture introduced in our previous publication. It demonstrates the capabilities of machine-learning-supported schedulability analysis under more realistic circumstances, which could be achieved by simulating functionalities present in a real automotive environment.

Both machine learning approaches achieved good results, but they are still far from perfect, especially considering the intended application in an automotive environment. We expect that a broader data set will enable further improvement of the machine learning models. It is relevant to note that, while both models showed a similar accuracy with a small task set, the LSTM net clearly outperformed the SNN when using a larger data set. We assumed two main reasons for this behavior, which we will explore in future work: the smaller data set resulted in overfitting, which could be further improved with a more robust data set, and the current network architecture is too simple. It is also relevant to mention that this is the first implementation, to our knowledge, of an SNN for schedulability analysis in a real-time system. This means that there is potential for improvement and that future performance could be improved with a more extensive data set, a more complex network architecture, and proper tuning of the neuron parameters. A further issue is the slow speed of the predictions, relative to the rest of the toolchain and also to the LSTM model. In order to reduce the impact of this issue, we plan to explore a few mitigations, such as further exploiting the toolchain feature of generating alternate distributions ahead of time by grouping their predictions. Other options are, for example, exploring the integration of a plugin that allows live input and output to the SpiNNaker (without having to re-upload

the SNN) or even the implementation of the SNN on a different neuromorphic hardware, such as Intel Loihi or IBM TrueNorth. However, it is also important to mention that an advantage of this approach over other machine learning techniques lies in the lower power consumption of networks implemented on SpiNNaker and other neuromorphic hardware when compared to conventional architectures, such as the one present in the Nvidia Jetson TX2 [31,32].

If we consider the accuracy results obtained, our machine learning models were outperformed in terms of accuracy by similar works, such as the ones demonstrated in [19] with 0.95 and in [9] with 0.955 using only machine learning and up to 0.99 when using a hybrid strategy combining a fallback manual calculation. Nevertheless, our approach is more flexible for applying to different architectures by using the generic architecture shown here. We also consider the amount of jobs for each task, as well as variable execution times, which might help obtain more realistic predictions, but at the cost of achieving a possibly higher false positive rate. As mentioned in [9], this rate should be kept as low as possible to avoid the deployment of infeasible task sets to the ECUs, so that is a necessary improvement to our models. Furthermore, we are considering that the setup could be extended with a fallback strategy when reaching higher criticality situations, where false positive predictions become more important.

In summary, the contribution in this paper can be divided into two main areas. First, we provided an insight into the deployment toolchain, which enables standard and relatively seamless tests for future developments of both the ECU systems or the machine learning approaches. Second, we demonstrated that schedulability analysis can be aided by using machine learning techniques, which we showed with two different networks. In particular, it is relevant to note that even with a first and simple approach for the implementation with the SNN, its results are comparable with those of the more complex LSTM network, while also having the potential of achieving better power efficiency. Therefore, we consider the developments described here an important milestone on our way to show how machine-learning-based schedulability analysis can enable task migration in an automotive environment, thus paving the way to efficient ECU consolidation. However, it is also important to mention that the setup is a first step approach at the moment, and given its importance to our work, we are planning on extending and improving its implementation in the future.

## 8. Conclusions and Future Work

The work documented in this paper provides a setup for testing different machine learning techniques and for enabling dynamic task migration among several ECUs. This is done by letting the ML predict the schedulability of the distributed task sets. In this paper, we also describe two approaches for performing the predictions using machine learning: an LSTM network running on a traditional architecture and a spiking neural network on a neuromorphic architecture. Furthermore, to our knowledge, it is also the first implementation of an SNN for this use case in this application domain. The integration of the two machine learning techniques with the setup and the ECU systems demonstrates the capabilities of the test setup. This work provides an important tool for our further research.

The next steps in our research involve the improvement of the machine learning algorithms described here, as well as the exploration of other algorithms. Furthermore, we plan to explore improvements to the ECU architecture to exploit its multi-core capabilities and integrate it with the setup.

**Author Contributions:** Conceptualization, B.B. and J.K.; Funding acquisition, J.K.; Investigation, O.D. and B.B.; Methodology, O.D. and B.B.; Project administration, J.K.; Software, O.D. and B.B.; Supervision, U.B.; Writing—original draft, O.D.; Writing—review & editing, O.D., B.B. and J.K. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research was partially funded by the European Union (EU) under RIA Grant No. 825050.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** Not applicable.

**Acknowledgments:** The tool described in Section 4 was initially designed and implemented with the support of Bernhard Blieninger's student Robert Hamsch as part of his Master's thesis. Further development of this work resulted in the approach described here.

**Conflicts of Interest:** The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

## References

1. Baunach, M.; Martins Gomes, R.; Malenko, M.; Mauroner, F.; Batista Ribeiro, L.; Scheipel, T. Smart mobility of the future—A challenge for embedded automotive systems. In *Proceedings of the e & i Elektrotechnik und Informationstechnik*; Springer: Berlin/Heidelberg, Germany, 2018; Volume 135, pp. 304–308. [CrossRef]
2. Hainz, C.; Chauhan, A. *Automotive Change Drivers for the Next Decade*; Technical Report; EY Global Automotive & Transportation Sector: London, UK, 2016.
3. Vipin, K.; Shreejith, S.; Fahmy, S.A.; Easwaran, A. Mapping Time-Critical Safety-Critical Cyber Physical Systems to Hybrid FPGAs. In Proceedings of the 2014 IEEE International Conference on Cyber-Physical Systems, Networks, and Applications, Nagoya, Japan, 6–7 October 2014; pp. 31–36. [CrossRef]
4. Vipin, K. CANNoC: An open-source NoC architecture for ECU consolidation. In Proceedings of the 2018 IEEE 61st International Midwest Symposium on Circuits and Systems (MWSCAS), Windsor, ON, Canada, 5–8 August 2018; pp. 940–943. [CrossRef]
5. Burkacky, O.; Deichmann, J.; Doll, G.; Knochenauer, C. *Rethinking Car Software and Electronics Architecture*; Technical Report; McKinsey & Company: Atlanta, GA, USA, 2018.
6. Sommer, S.; Camek, A.; Becker, K.; Buckl, C.; Zirkler, A.; Fiege, L.; Armbruster, M.; Spiegelberg, G.; Knoll, A. RACE: A Centralized Platform Computer Based Architecture for Automotive Applications. In Proceedings of the 2013 IEEE International Electric Vehicle Conference (IEVC), Silicon Valley, CA, USA, 23–25 October 2013; pp. 1–6. [CrossRef]
7. Shankar, A. *Future Automotive E/E Architecture*; IEEE India Info.: Banglore, India, 2019; pp. 68–73.
8. Buttazzo, G.C. *Hard Real-Time Computing Systems*; Springer Science+Business Media, LLC: New York, NY, USA, 2011. [CrossRef]
9. Mai, T.L.; Navet, N.; Migge, J. A Hybrid Machine Learning and Schedulability Analysis Method for the Verification of TSN Networks. In Proceedings of the 2019 15th IEEE International Workshop on Factory Communication Systems (WFCS), Sundsvall, Sweden, 27–29 May 2019; pp. 1–8. [CrossRef]
10. Delgadillo, O.; Blieninger, B.; Kuhn, J.; Baumgarten, U. An Architecture to Enable Machine-Learning-Based Task Migration for Multi-Core Real-Time Systems. In Proceedings of the 2021 IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC), Singapore, 20–23 December 2021; pp. 405–412. [CrossRef]
11. Megel, T.; Sirdey, R.; David, V. Minimizing Task Preemptions and Migrations in Multiprocessor Optimal Real-Time Schedules. In Proceedings of the 2010 31st IEEE Real-Time Systems Symposium, San Diego, CA, USA, 30 November–3 December 2010; pp. 37–46. [CrossRef]
12. Faizan, M.; Pillai, A.S. Dynamic Task Allocation and Scheduling for Multicore Electronics Control Unit (ECU). In Proceedings of the 2019 3rd International conference on Electronics, Communication and Aerospace Technology (ICECA), Coimbatore, India, 12–14 June 2019; pp. 821–826. [CrossRef]
13. Chen, Y.Y.; Lyu, C.M. ECU-level fault-tolerant framework for safety-critical FlexRay network systems. In Proceedings of the 2012 International Conference on ICT Convergence (ICTC), Jeju, Korea, 15–17 October 2012; pp. 553–558. [CrossRef]
14. Lee, J.; Shin, S.Y.; Nejati, S.; Briand, L.C.; Parache, Y.I. Schedulability Analysis of Real-Time Systems with Uncertain Worst-Case Execution Times. *arXiv* **2020**, arXiv:2007.10490.
15. Aradhya, S.; Thejaswini, S.; Nagaveni, V. Multicore Embedded Worst-Case Task Design Issues and Analysis Using Machine Learning Logic. In *Proceedings of the IOT with Smart Systems*; Senjyu, T., Mahalle, P., Perumal, T., Joshi, A., Eds.; Springer: Singapore, 2022; pp. 531–540.
16. Cardeira, C.; Mammeri, Z. Neural networks for multiprocessor real-time scheduling. In Proceedings of the Sixth Euromicro Workshop on Real-Time Systems, Vaesteraas, Sweden, 15–17 June 1994; pp. 59–64.
17. Domínguez, E.; Jerez, J.; Llopis, L.; Morante, A. RealNet: A neural network architecture for real-time systems scheduling. *Neural Comput. Appl.* **2004**, *13*, 281–287. [CrossRef]
18. Guo, Z.; Baruah, S.K. A Neurodynamic Approach for Real-Time Scheduling via Maximizing Piecewise Linear Utility. *IEEE Trans. Neural Netw. Learn. Syst.* **2016**, *27*, 238–248. [CrossRef] [PubMed]
19. Hoffmann, J.L.C.; Fröhlich, A.A. Online Machine Learning for Energy-Aware Multicore Real-Time Embedded Systems. *IEEE Trans. Comput.* **2022**, *71*, 493–505. [CrossRef]
20. De Bock, Y.; Altmeyer, S.; Broeckhove, J.; Hellinckx, P. Task-Set generator for schedulability analysis using the TACLeBench benchmark suite. In Proceedings of the Embedded Operating Systems Workshop: EWiLi 2016, Pittsburgh, PA, USA, 6 October 2016.
21. Navet, N.; Mai, T.L.; Migge, J. *Using Machine Learning to Speed Up the Design Space Exploration of Ethernet TSN Networks*; Technical Report; University of Luxembourg: Luxembourg, 2019.

22. Maruf, M.A.; Azim, A. Extending resources for avoiding overloads of mixed-criticality tasks in cyber-physical systems. *Iet-Cyber-Phys. Syst. Theory Appl.* **2020**, *5*, 60–70. [CrossRef]
23. Orhean, A.I.; Pop, F.; Raicu, I. New scheduling approach using reinforcement learning for heterogeneous distributed systems. *Proc. J. Parallel Distrib. Comput.* **2017**, *117*, 292–302. [CrossRef]
24. Falk, H.; Altmeyer, S.; Hellinckx, P.; Lisper, B.; Puffitsch, W.; Rochange, C.; Schoeberl, M.; Sørensen, R.B.; Wägemann, P.; Wegener, S. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In *Proceedings of the 16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*; Schoeberl, M., Ed.; Schloss Dagstuhl–Leibniz-Zentrum für Informatik: Dagstuhl, Germany, 2016; Volume 55, pp. 2:1–2:10.
25. Sha, L.; Abdelzaher, T.; Arzen, K.E.; Cervin, A.; Baker, T.; Burns, A.; Buttazzo, G.; Caccamo, M.; Lehoczky, J.; Mok, A.K. Real Time Scheduling Theory: A Historical Perspective. *Proc. Real-Time Syst.* **2004**, *28*, 101–155. [CrossRef]
26. Buttazzo, G.C. Rate Monotonic vs. EDF: Judgement Day. *Proc. Real-Time Syst.* **2005**, *29*, 5–26. [CrossRef]
27. Painkras, E.; Plana, L.A.; Garside, J.; Temple, S.; Davidson, S.; Pepper, J.; Clark, D.; Patterson, C.; Furber, S. SpiNNaker: A multi-core System-on-Chip for massively-parallel neural net simulation. In Proceedings of the IEEE 2012 Custom Integrated Circuits Conference, San Jose, CA, USA, 9–12 September 2012; pp. 1–4. [CrossRef]
28. Roy, K.; Jaiswal, A.; Panda, P. Towards spike-based machine intelligence with neuromorphic computing. *Nature* **2019**, *575*, 607–617. [CrossRef] [PubMed]
29. Upadhyay, N.K.; Joshi, S.; Yang, J.J. Synaptic electronics and neuromorphic computing. *Sci. China Inf. Sci.* **2016**, *59*, 1–26. [CrossRef]
30. Gerstner, W. Spiking Neurons. In *Pulsed Neural Networks*; Maass, W., Bishop, C.M., Eds.; MIT Press: Cambridge, MA, USA, 1998; Chapter 1, pp. 3–53.
31. Paolucci, P.; Ammendola, R.; Biagioni, A.; Frezza, O.; Lo Cicero, F.; Lonardo, A.; Martinelli, M.; Pastorelli, E.; Simula, F.; Vicini, P. Power, Energy and Speed of Embedded and Server Multi-Cores applied to Distributed Simulation of Spiking Neural Networks: ARM in NVIDIA Tegra vs. Intel Xeon quad-cores. *arXiv* **2015**, arXiv:1505.03015.
32. Stromatias, E.; Galluppi, F.; Patterson, C.; Furber, S. Power analysis of large-scale, real-time neural networks on SpiNNaker. In Proceedings of the 2013 International Joint Conference on Neural Networks (IJCNN), Dallas, TX, USA, 4–9 August 2013; pp. 1–8. [CrossRef]