Logo Placeholder, see README

Fakultät für Informatik
Technische Universität München

Logo Placeholder, see README

Integration of Mixed-Criticality into Dynamic Task Migration for Real-Time Embedded Systems

Octavio Ivan Delgadillo Ruiz

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

PhD (check specific degree))

genehmigten Dissertation.

Vorsitzender:
Prof. Dr. Uwe Baumgarten

Prüfende der Dissertation:
1. Prof. Dr.-Ing. Vorname Nachname
2. Prof. Dr.-Ing. Vorname Nachname

Die Dissertation wurde am 01.01.2016 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 10.05.2016 angenommen.

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Zusammenfassung

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Contents

List of Figures

List of Tables

1 Introduction

**** FIX BIBLIOGRAPHY, not loading for some reason *****

Trends in the automotive industry are shaping the future of cars in a way that electronic and computing devices are becoming increasingly important. In fact, the majority of innovations in the automotive sector are related to electronic systems, either in the form of hardware or software [HC16]. Developments such as vehicle electrification, autonomous driving and vehicle connectivity are only some examples of automotive applications where computer processing has a big relevance [KSK18]. It is therefore reasonable to seek for approaches to ensure their usage is efficient and safety requirements are met.

In modern cars, dozens of computers, commonly known as electronic control units (ECUs), execute the various computing tasks present in a car. This number is likely to increase if we consider past trends: as the tasks performed by ECUs often have safety-critical constraints, such as real-time capabilities, they are commonly integrated to serve a specific purpose, avoiding conflicts caused by the parallel execution of other tasks [Vip+14; Vip18]. However, this strategy has shortcomings in the form of inefficient usage of the electronic devices (many tasks are only executed in specific and rare situations), reduced fault tolerance if a system fails, and increased weight and cost of the vehicle due to the high number of ECUs and cables [Vip18; Bau+18]. Hence, it is an important topic in the automotive industry to find solutions for these issues, especially optimizing costs, while ensuring vehicle safety is kept [Bur+18].

ECU consolidation is an approach for the reduction in the number of electronic devices in the car. The idea is to consolidate the execution of the tasks from many single-purpose ECUs to a few powerful, multi-purpose ECUs. However, the implementation of ECU consolidation raises other challenges: as more tasks need to be executed on the same platform, higher computation and safety requirements must be met. Regarding safety, it is important to consider the added complexity, and cases where a hardware or software failure occur need to be considered, since a single failure could block many tasks and cause major issues. Also, some safety-critical tasks require redundancy to ensure their correct execution [Mun+17]. These challenges have motivated previous research at the chair of operating systems of the Technical University of Munich, where research on projects such as KIA4SM and MaLSAMi has explored the concept of dynamic task migration as a process where the execution of any tasks could move at any given time from one ECU to another. This would allow tasks to finish execution in case an ECU is overloaded or stops working, effectively allowing them to finish executing and meet their real-time constraints.

Previous work at the chair has divided the task migration process in two stages: planning and execution. The planning is the stage which generates a task distribution to the devices that will execute them. The execution is then responsible for allowing tasks to migrate from one device to another, while ensuring their progress is not lost and the execution continues and finishes correctly at the target device. So far, the work performed at the chair has explored the approach with a single criticality level, using different priority strategies for scheduling the tasks on an ECU (that is, deciding which task should get the processor and be executed at a given time). While this idea is in itself an important contribution, it is relevant to note that in many safety-critical embedded applications, such as the aerospace and automotive industries, different criticality levels exist, which are often defined in standards such as the ISO 26262, which defines the ASIL (automotive safety integrity level). For example, in a car, the correct functioning of the ABS is highly critical, as a failure could be fatal. In contrast, the functionality of the radio system has a lower criticality, as an eventual failure would not cause any issue other than an unpleasant trip. This idea is also important because in these industries there exist certification agencies that validate a system before it can be distributed.

For this reason, and due to the relevance of the concept of mixed-criticality in many industries, it would be important to expand the research on task migration to explore its integration. In particular, expanding the execution stage to ensure tasks with higher criticality are always able to execute properly and meet their deadline is crucial. But also it is important to find a balance between resource efficiency and reliability of the migration process for different criticality levels. Therefore, the work proposed as part of this PhD program aims to implement a strategy for the migration execution at different levels of criticality. Additionally, the mixed-criticality concept has to be integrated as well into the planning stage. This will be also explored in the scope of this research, although the main focus will be in the execution stage.

Furthermore, another important trend is the integration of edge technologies (known as multiaccess edge cloud or mobile edge computing - MEC - or in the specific automotive use-case vehicular edge cloud - VEC-) for augmenting the vehicle's capabilities in aspects such as communication with other traffic participants or information services, increased sensing capabilities, offloading of tasks, etc (add reference). In the scope of this dissertation, it is considered that edge computing resources can be used to provide a further execution possibility for non-critical tasks under 2 circumstances: 1) the task would benefit from the availability of more powerful computing resources and sensing inputs, and 2) the physical hardware is at a critically high task load level and low criticality tasks would otherwise starve. It is considered that for the most highly critical tasks (like the ones interacting directly with the control of actuators like steering or vehicle speed) this approach is not feasible, as communication with an external component is always prone to interruption, even with the low latencies provided by new communication technologies such as 6G or 5G (add ref).

Although the work performed as part of this research is based on previous developments at the chair and will likely extend existent tools, it includes the development of a platform that implements mixed-criticality scheduling, as well as a migration system that involves the concept in different areas in the planning and execution stages. This includes the selection of hardware and software, as well as network interfaces. Also, verification of the algorithms used will be performed to ensure safety and timing requirements are met.

Research Questions

To better define the scope and goal of the research performed in this thesis, following research questions have been propsed as basis (*** Maybe remove and integrate better into Intro text****):

- 1. To what extent is task migration feasible in such a distributed real-time system while respecting real-time constraints, and under what conditions?
- 2. Given the system's need to handle uncertainty and adapt to changes in hardware availability and task load, how should tasks be prioritized based on criticality?
- 3. What migration approaches - both at communication level and at OS level - are most suitable for high- and low-criticality tasks considering a trade-off between minimizing downtime and optimizing resources?
- 4. How feasible is extending the system by offloading tasks to virtual edge resources? What are the implications of this approach, especially in terms of latency, response time, safety and resource optimization?

—- See how to integrate into text but: Following chapters / aspects help find an answer to the research questions: Q 1: ... Q 2: Approach, Base OS Q 3: Migration, Q 4: Base OS, Results

Related Work

At the chair, previous research projects, such as KIA4SM and MaLSAMi, have explored the possibility of migrating tasks running on a device to another in a real-time capable system (for example, ECUs in a vehicle) under certain conditions. As mentioned before, in these works, the migration was divided into two main stages: The first is the migration planning, which determines the hardware that tasks will be migrated to, should the original hardware not be able to fulfill its duty (for example, if there is a failure in that hardware or if the real-time constraints or deadlines would be violated).

The second is the execution of the migration, which ensures that corresponding tasks can be migrated from the source device to the target device while keeping their current state.

The migration execution has been explored previously at the chair, for example in projects KIA4SM and HaCRoM. This was explored in the form of a real-time checkpoint-restore mechanism. This involves creating and storing a snapshot of running tasks in a shared memory or copying the memory from a device to another. The work is based on Fiasco.OC and Genode OS.

MaLSAMi and subsequent theses analyzed migration planning, with researchers performing schedulability analysis based on machine learning (specifically, on neural networks). Machine learning was picked over traditional mathematical approaches such as the ones proposed by Buttazzo [But11], because the recurrent calculations can become too complex for complex tasks and for big task sets, and they often lead to a pessimistic calculation of the system utilization. By predicting the feasibility of a task set using machine learning algorithms, potentially faster but less precise results are obtained, as demonstrated by previous theses by Taieb [Tai19], Utz [Utz19] and Blieninger [Bli19]. The predictions provided by the machine-learning approach indicate whether a task set is 100% schedulable or not, but they are not completely safe, since false positive predictions may occur. This approach could be a potentially powerful solution for enabling the execution at run-time of the real-time capable migration planning.

Additionally, a few different platforms and setups have been explored in these projects. The used operating systems running on the ECUs are Genode OS, as used in MaLSAMi and a few theses, a real-time operating system based on an extension of Genode OS with Fiasco.OC, as used in KIA4SM [EKB15] and HaCRoM, and FreeRTOS, as used by Delgadillo [Del21]. These developments are considered in the selection of the platform.

These projects have achieved research-relevant results in their segments, but the concept of mixed-criticality has not been explored in related chair internal work. It is therefore necessary to look at relevant chair-external work. In particular, those regarding mixed-criticality scheduling strategies and related to task migration are reviewed next.

It is worth noting that to my knowledge, research on mixed-criticality task migration of high criticality tasks, especially applied to the idea of ECU consolidation, has not been published. A possible reason for the lack of research in this area is the fact that nowadays safety is valued much higher than resource efficiency, thus overseeing the potential for optimization in this aspect. However, in my opinion, it should be possible to achieve both goals by implementing a holistic strategy and therefore it is an area where research can contribute importantly.

2 State of the Art

*** NEED TO ORGANIZE BETTER MY LITERATURE AND RELATED WORK ***

2.1 Automotive Embedded Devices / Multicore

- Add reference of current automotive architectures / AUTOSAR. Explain relevance of multicore approaches - AMP vs SMP
- Add references related to traditional hard real-time scheduling – not considering Mixed Criticality. (Davis & Burns, Buttazzo)

2.2 Real-Time Distributed Systems

-

2.3 Mixed Criticality Scheduling

Until last decade, the concept of mixed criticality and its involvement in the scheduling of tasks has been mostly explored for single processor systems. In 2011, Baruah et al. proposed an adaptive mixed-criticality scheduling algorithm which set different WCET for different criticality levels of the same task, changing to a higher criticality mode if a job would not report completion after its low criticality deadline [BBD11]. Then, also in 2011, Baruah et al. proposed a mixed-criticality scheduling algorithm called EDF-VD for a single processor, which introduces a high criticality mode that modifies high criticality task deadlines to give them a higher priority over low criticality task [Bar+11]. This algorithm is defined for any number of criticality levels. In 2012 Li and Baruah proposed an extension of the EDF-VD scheduling algorithm to multiprocessors by adding fpEDF strategy, which adds higher priority to tasks with utilization values higher than 0.5 [LB12]. Their approach did not explore migration strategies, but was one of the first implementations of a mixed-criticality scheduler on a multiple processor system. In 2015, Gratia et al. adapted RUN, a global multiprocessor scheduling algorithm, to support mixed criticality [GRP15]. This approach also considers only a dual criticality system, and only considers the migration of low criticality tasks.

2.4 Task Migration (considering Mixed Criticality)

In 2018, Ramanathan and Easwaran investigated an approach for mixed-criticality scheduling on multiple processors that involved partial migration based on fixed partitions for some tasks and leaving some low criticality tasks free to execute on any available resource [RE18]. In 2019, Zeng et al. tried a similar approach [ZLL19]. However, their approaches only considered dual-criticality; alsoonly low criticality tasks would migrate from one processor to another, to ensure these tasks also get execution time, even when processors enter high criticality mode and focus on finishing high criticality jobs. In the case of high criticality tasks, they were mapped statically to a single processor. ¨

2.5 Vehicular Edge Cloud

- Add reference of survey study by Raza et al.
- Add reference to Mark8s as base ofr the containerized approach
- Add reference of cloud related task migration (some in Ruano, some in paper list)
- Add reference of current MEC technology / 5G / 6G and communication latencies. This will be important to justify assumptions in the process for migration to virtual devices

*** KEEP FOR REFERENCE commands...*** Example citations [**barham2003xen**; **LIS**]. Example acronym usage Central Processing Unit (CPU).

3 Methodology and Concepts

4 Approach

The proposed system is a "breathable" distributed system, where breathable means that it can react at runtime to distinct situations to reconfigure itself and update a decision of where the tasks will execute, always prioritizing the execution of tasks based on their criticality. The situations that trigger this breathable behavior are mainly categorized in two types: 1) changes to the set of tasks that the system must execute (task queue) and 2) changes to the availability of computing resources. In the first category, it is assumed that all tasks have at least soft real-time behavior and they can all be categorized as periodic (need to be executed every certain time period) or sporadic (are executed upon request and at unpredictable moments) [**add reference**]. In the scope of this work and for simplicity, we assume two generalizations: a) all tasks can be treated as periodic tasks with a variable number of jobs (individual executions of the task that can be repeated), where specific cases with 1 total job represent sporadic tasks and an infinite number of jobs represent common periodic tasks, which could be split in batches of jobs for less critical tasks; and b) all tasks can be migrated from one device to another in between jobs. Furthermore, in the scope of this work it is assumed that tasks have no data or sequential dependencies with each other. The second category includes reacting to addition or removal of physical devices or hardware (for example, due to hardware failure or device restarts triggered by updates, etc.) and also to the availability and latency of virtual devices in the mobile-edge-cloud (for example, when driving over a highway section with MEC support).

These characteristics of the system can be summarized in the image below (add proper reference). The system behavior can be then described as follows. First, a reconfiguring situation triggers a re-distribution of the tasks that need to be executed, which are kept in the global queue. This global task queue as well as the set of both physical and virtual devices (or ECUs) are fed to a task partitioning algorithm, which decides a task-to-board mapping and then triggers the distribution of the tasks on the respective mapped device. The final step is that, depending on the previous mapping of tasks to boards, if the assigned ECU for a task changes, it is necessary to migrate its execution from the previous device (source) to the new one (target). This last step is the main objective of this research work, and in the scope of it, a few requirements should be considered to deem the implementation as successful: 1) the task has to be stopped in the source device and started in the target device, 2) the relevant task context should be also migrated, meaning that no crucial data is lost (in the case of a SLAM algorithm, the map and position would not be lost), 3) both the migration decision and execution strategies should consider the criticality of the tasks, 4) real-time behavior

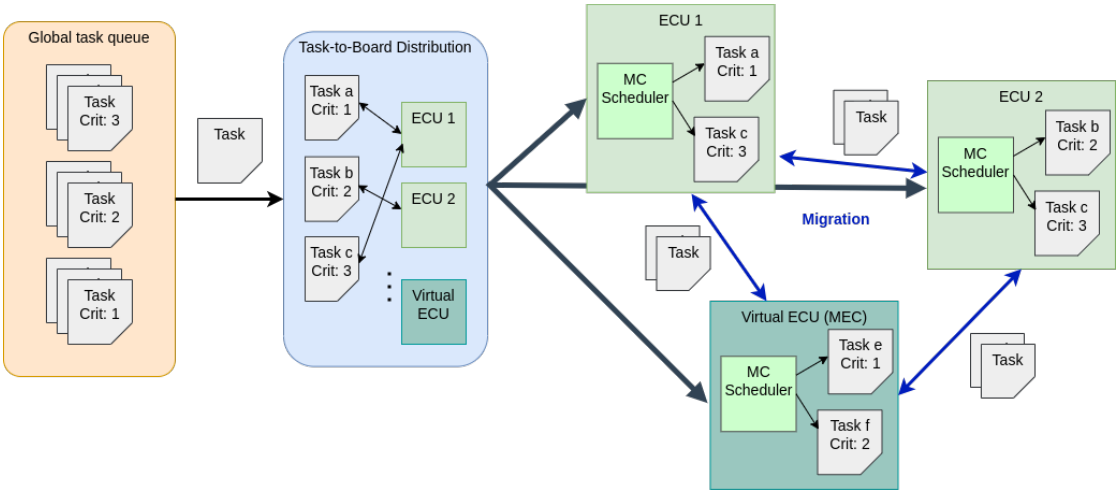

Figure 4.1: General System Overview

should be also considered in the migration strategy. It is relevant to mention that a master ECU would be responsible for both keeping the global task queue, triggering the task-to-board distribution algorithm (which itself can run on a separate ECU or a specialized ML hardware) and keeping the relevant files to start tasks when necessary.

Based on the previous list of requirements, and especially 3 and 4, at least 2 migration strategies should be implemented, since tasks with different criticalities also have different constraints to be prioritized. For example, in the case of low-criticality tasks, it might be more relevant that the task does not end in starvation, whereas for a high critical task it is crucial that the result is delivered always on-time and that, if the task needs to be paused due to a critical failure, it is brought back up in a minimal and deterministic time to keep up with the real-time constraints.

4.0.1 Contributions

This is just conceptually speaking... But in order to better place my contribution

- Task-to-Board distribution/migration planning: Toolchain for integrating ML was result of Bernhard + Robert Hamsch's thesis. My contribution: adaption to my embedded devices + include ECU, adaption to send ELF files + checkpoints, adaption to keep PTP synchronization
- ECU base system (OS): Selection of FreeRTOS as base RTOS (Bernhard, probably with input from Oliver), confirmed by me. Decision of AMP over SMP in multicore case (Bernhard + me). Some inter-processor features (probably Oliver + students). My contribution: extension of ESFREE to include MC (supported by students Mahdi and Kamran). Conceptual architecture of ELF loader, wrapper task, checkpoint keeping (implementation Yinbo).

- Guest task architecture: My contribution (supported by results of Yinbo): C task architecture in form of a library + simplified linker script (adapted from Yinbo's thesis)
- Migration strategy HI: My contribution: concept definition. Implementation partially shown in PoC by Dorota, must be extended still + maybe will require distributed shared memory implementation?
- Migration strategy LO: Concept definition (me) + implementation and PoC by Yinbo
- Migration strategy Virtual ECU (LO): Concept definition (me + Bernhard) + partial implementation and PoC analysis by Roberto + adaption and integration (me + probably Mahdi)
- Further system features: PTP + SOMEIP (Yinbo implementation + me/Bernhard concept), strategy for system functions (me), missing - network, other peripherals, interrupts, mem management?

4.1 General Overview

As mentioned before, the main objective of the proposed research is to achieve the integration of mixed criticality into the task migration process. The proper integration would require different strategies for the different criticality modes. For this reason, it is first important to define what is understood under mixed criticality in the scope of this work. While many publications consider only two criticality levels (high and low), the standards commonly describe several of them. In this work, the concept of multiple criticality levels will be used in a conceptual phase following the ASIL standard in ISO 26262, while the implementation and results will be collected for at least 2 levels in both the scheduling algorithm and the migration strategy. Whereas a model with multiple criticality levels represents real situations better over one with just two, as tasks can have intermediate criticality levels, in the scope of this work, the focus relies more on fulfilling challenges at both ends of the criticality spectrum, and assuming a mix of both strategies might apply as a solution for intermediate levels. One concept that should be covered is the consideration of different execution times according to a criticality level, as normally higher criticality levels offer pessimistic WCET estimates to ensure correct behavior occurs, but actual execution times are often lower. This is a concept that is explored in many publications and is often necessary to ensure compliance with certification authorities.

The first element needed for the system developed is the integration of a mixed-criticality scheduler for each of the physical devices. This should ensure that for any given device, the tasks with a higher criticality will never miss the deadline for a job,

while being more flexible for lower criticality levels. In this regard, exploring different mixed-criticality scheduling algorithms and their integration in the development platform is an important step that will allow the migration to occur and be evaluated properly. Several publications have proposed mixed-criticality scheduling algorithms, such as [Bar+11; Fle13; ZWL17; BBD11; Li+13], and a few of them can be implemented and compared. It is also important to mention that following the state of the art in embedded devices, a multi-core implementation is desired, with asymmetrical multiprocessing (AMP) being the preferred solution, as software failures could be isolated to affect less tasks (***find a reference***). In particular, following approaches are explored (so far): hierarchical virtual deadline EDF (faulty implementation); reserved cores for 2-level MC, with LO crit cores as backup cores for HI tasks. It is assumed that this step should not affect the behavior of the virtual devices, since by concept, they can only run non highly (not necessarily only LO) critical tasks, due to the uncertainties introduced by the external communication with the MEC (explain further...).

Another important element for the distributed system to ensure its compliance to the real-time constraints is a timing protocol such as PTP, to keep the different devices synchronized and share the same notion of time when following task deadlines. This way, even when migrating a task, information on the deadlines should be very accurate. In addition to this, migration strategies must consider that transmission times can be variable and, in the ideal case, the source device should keep the original task execution until it is ensured that the target device has started the execution successfully, potentially taking longer than the deadline of a running job.

The migration planning will be adapted from previous work, namely the student theses supervised by Bernhard Blieninger. As these don't consider mixed-criticality or a multi-core implementation in the system behavior, a few changes are necessary to integrate both concepts. First, the schedulability analysis that is commonly used for selecting the best task distribution has to be extended to work with criticality levels and the use of a mixed-criticality scheduler. Here, it has to be taken into consideration whether extending and retraining the machine learning models is worth the effort, especially since this idea is still not proven as a solution and the extension with the criticality levels and other relevant information might add complexity to the net. Otherwise, implementing a mathematical schedulability analysis that considers mixed criticality is necessary. A second extension that should be performed in the planning stage is to penalize the migration of higher criticality tasks, as the migration overhead adds an additional risk for the tasks failing. Furthermore, the availability of both virtual and physical devices needs to be considered at the moment of deciding which tasks should migrate and where.

The next step in the development of this strategy is the execution of the migration, which is the core concept explored in this work. It is important to consider the necessity to meet strict timing constraints for tasks at the higher end of the criticality spectrum. ***THIS HI criticality solution is yet to be explored, Yinbo is working on the variant for

LO crit*** While the exact approach is yet to be proven as feasible, an idea for higher criticality tasks would be to let them run in a standby state in all or a subset of the total of available ECUs, and only executing it in one ECU, while storing runtime generated data either in a central unit or in a shared memory only accessible to highly critical tasks, in this way only a start/stop signal is sent to the task and the execution can be resumed quickly, also ensuring fail-safety in case the executing device fails. Also, for tasks that require redundancy (e.g. ASIL level D), it should be possible for tasks to execute actively multiple times and produce results in parallel. Another addition in this stage should be the introduction of a real-time capable communication protocol. In this sense, it should be possible to bound the migration time for this tasks, ideally in the range of a few milliseconds. These ideas could be faster to execute and allow for less variance in the time, eventually making it possible to perform formal verification, but it would be expensive if tasks at all criticality levels were to run like that, as the resources are utilized inefficiently. This should be acceptable for highly critical tasks, though, since the majority of the tasks would be assigned low to medium criticality levels.

The strategy for migrating lower criticality tasks between physical devices is prone to more flexibility, so that a wider range of tasks can be deployed with a lower impact in terms of resource efficiency, even if this would open the possibility for more erroneous behavior in the tasks. The approach currently explored for tasks with low criticality involves the transmission of precompiled task binaries and a snapshot of the execution data every time a task is distributed to an ECU and the usage of a normal TCP/IP protocol over Ethernet. ***This is Yinbo's ELF loader thesis*** With this strategy, the resource utilization in the devices is made more efficient, but the time spent for the migration is increased as the communication is slower due to the amount of information exchanged, and also less reliable due to the nature of the communication protocol.

As an optional step, the exploration of strategies for more criticality levels is yet to be done, but a few ideas are suggested. First, a combination of the two strategies mentioned could be implemented for intermediate criticality levels, such as leaving the tasks running but using a less predictable communication protocol. Another idea is to subdivide each of the criticality levels mentioned and there perform variations of the mentioned strategy. For example, in a system with a certain number of devices and 2 high criticality sublevels, the highest sublevel tasks are kept in standby on all devices, while the rest are only kept in standby in a few of the devices, making sure there is always an ECU ready for the highest criticality and using less resources for the second sublevel. In the case of lower criticality, this could be implemented in the form of giving priority to the migration and transmission of data of tasks at the higher sublevels.

Additionally, due to the consideration of virtual devices in the MEC, an additional strategy is also implemented in the scope of this work. That is, the migration of low criticality tasks from and to the MEC. For this implementation, it is assumed that the

MEC servers share some aspects of the architecture with the real hardware and that the devices can be modeled in the form of containers as their digital twins. In this way, upon availability, the master ECU can request to start a container which will run the required task. It is assumed that this behavior is allowed for low criticality tasks that can benefit from the more powerful resources available in the MEC, and that communication latency can be neglected as long as the results produced are relevant. An example of this might be a path planning algorithm for an autonomous vehicle, where the immediate safety of the vehicle is not threatened, but the algorithm can be extended with further sensor data and better hardware for performing ML tasks. ***Part of this has already been implemented by Roberto*** It is relevant in the scope of the migration, that the virtual tasks have an equivalent task implemented for the real devices, and that the task data can also be migrated between them. In this way, the task can be moved depending on the need and availability of the resources to and from the MEC.

4.2 Overall System Architecture

- Master ECU responsible for keeping track of global queue list and available hardware. Also responsible for triggering task-to-board distribution and deployment. This master ECU might have a backup ECU to meet requirements and furthermore keeps all relevant data for all tasks (precompiled task binaries / container images, context data files, criticality levels). Current implementation represents master ECU with a more powerful workstation - could be implemented to work on an embedded board similar to the other devices.
- Component performing task partitioning algorithm (might be ML-supported schedulability analysis as developed with Bernhard). Assumed to be a "black box" in most of the thesis as it is not in the scope of this research.
- Physical ECUs based on a FreeRTOS kernel extended with the functionality needed for the task migration and to comply with MC, as well as resource optimization through multicore. Some of these ECUs might be kept in a hot-standby mode or off to have backup devices while minimizing energy consumption in the vehicle
- Virtual ECUs based on a containerized minimal version of the FreeRTOS system that can execute the same tasks without the need for changes in the code
- Embedded OS Software Architecture: Extensions to the FreeRTOS Kernel - scheduler, deployment slave, inter-core communication, execution and monitoring of real-time behavior
- Task Specific Software Architecture: To be able to execute tasks with the system, developers should consider a few things when developing new tasks. For example:

definition of context data structure, loading and storing of context, external function handles, handling of system functions both in physical and virtual device...

4.3 Operating System / FreeRTOS implementation + internal behavior

Should cover following bullet points into detail when implementation is complete

- AMP multicore implementation on the Ultra96's quad-core ARM Cortex A53 (real hardware). Core 0 acts as the "master" core and cores 1-3 as "slave" cores. Core 0 interacts with master ECU and distributes tasks assigned to the board internally to the cores, following the MC (mixed-criticality) approach.
- To ensure tasks are prioritized according to criticality, MC is implemented also in the base behavior of the embedded devices. Two variants on MC implementation: 1) reserved cores with core-level EDF (avoiding conflicts between criticality levels) and 2) hierarchical virtual EDF
- Each core has "management" tasks (threads) to ensure proper functioning and dynamic functionalities: communication with master ECU (only master core), inter-core communication, task control and monitoring, and a scheduler thread
- Communication thread responsible for communicating with other cores and master ECU (only for core 0). This includes receiving instructions to execute/stop tasks, receiving and sending back task binaries (ELF) and data files, and sending back runtime data on the status of the executed tasks and the device's health.
- Task control and monitoring task involves: starting and stopping of tasks, creating new threads for the executed tasks, loading their executables, passing the stored task context to the task or storing it after task execution, collecting runtime data for the task (number of successful jobs, deadlines missed, last execution)
- Scheduler thread responsible for assigning dynamic execution priorities to tasks according to both real-time constraints and criticality (except in reserved cores approach)
- Virtual device implementation through a single FreeRTOS container running on ARM based server. Base system is a twin of the real device, but compiled for the actual processor in the server (as of now not possible to run same ELF on different ARM platform or inside the container... will explore if possible). Currently container executes task directly (without management overhead). If ELF can run on the container, it might make sense to first boot a clone with all the management capabilities and then load the required ELF inside the container

- System tested with simulated workload with different tasks. Two types are necessary: 1) dummy tasks allow us to test high workloads by making the CPU work without the implementation effort for different tasks, 2) actual data processing tasks allow us to test the system functionality for the migration and to demonstrate it in a tangible way
- Dummy tasks are implemented by adapting COBRA framework as it has demonstrated in the scope of my thesis to be an easy way of producing different workloads with variable real-time constraints
- Actual tasks have yet to be implemented and tested, but some ideas: image processing (e.g.: edge detection), SLAM with LIDAR... These may need to additionally deal with inputs/outputs

4.4 Task-Device Partitioning / Migration Planning

The main goal of this work is to enable the breathability of the system by making it possible to execute any task on any available virtual or physical hardware resource. However, this requires a runtime component taking decisions on where the tasks should be executed at a point in time. To achieve this, the methods explored in [**cite migration planning literature / work from Bernhard, Hamsch, etc.**] are integrated into the test setup, along with simulated task loads and hardware changes (as described in the use cases / scenarios) that will trigger the migration of tasks from one device to another.

4.5 Internal Communication

The cores communicate with each other through inter-processor interrupts and ring buffers in an allocated shared memory space, allowing for the implementation of message queues for each core. Through the implementation of "post office" library [**explain behavior and refer to Horst/Wiesboeck**], each core has a thread handling the receiving of messages in the queue, allowing the actual management task to process the message further. In general, communication between any two cores of the processor is possible using this method, but most inter-core communications in this implementation occur between the master core and a slave core.

4.6 External Communication

- Communication with other devices enabled through LWIP (lightweight IP) library, offering basic functionality: IPv4 and v6, TCP and UDP sockets, DHCP...
- Communication with master ECU through periodic TCP messages with specific format (for now using JSON)

- Transmission of files (especially ELF and context) with an implementation of SOME/IP
- Current limitation: only core 0 has control over the LWIP stack. Need to enable access for user applications from the other cores

4.7 Use Cases / Test Cases

- 1. Physical Device Failure: Tasks with HI crit have a backup device ready to continue execution, minimizing downtime. Tasks with LO crit in both backup device and faulty device migrate to available hardware. In a situation where available hardware is not able to execute all tasks in global queue, HI crit tasks are prioritized and LO crit tasks might starve or skip jobs until workload is lower or additional hardware is available again.
- 2. Increase/Decrease of Task Load: Tasks might need to be redistributed and eventually backup hardware made active. This redistribution of tasks means that migration of running tasks is likely to be triggered to achieve a stable configuration where all tasks can execute properly. In this case, migration of highly critical tasks should be avoided, except for extreme cases, as they could miss crucial deadlines or uncertainty could be introduced into their calculations and response time. In the case of a decrease in the task load, it is likely that for energy optimization, a device can be shut down, or that the task load is balanced to ensure all devices run stably.
- 3. MEC is available and a task can be supported by MEC: In this case, a task that might benefit from further resources in the MEC (like a task performing heavy ML algorithms or a task that can improve results by sensor fusion including further cameras and other information stored in the edge) is migrated to the MEC. This task can have a counterpart running on the physical hardware to ensure basic functionality is kept while enhanced functionality is integrated as long as the MEC resources are available. An example of this can be a path planning algorithm for an autonomous vehicle, where the basic functionality might be following safely a road, while the enhanced functionality might find alternative routes depending on the traffic or accident report along the road.
- 4. MEC is available and task load is critically high on ECU devices, for example, due to hardware failure (like in use case 1). In this case, as low criticality tasks would start going into starvation, they can be offloaded temporarily to the MEC, still getting time to execute, although probably introducing some latency due to external communication with the vehicle. One such case might be an infotainment service, such as streaming. In this case, it is likely that an intermediate task would have to forward the task results into the vehicle network or any resources needed.

5 Base Operating System: Axcan OS

To enable the proposed breathable distributed real-time system — capable of dynamic task migration across heterogeneous embedded nodes — a shared, minimal, and portable operating system stack is essential. This stack must satisfy several core requirements: it must be lightweight enough to operate on constrained electronic control units (ECUs) and virtualized edge platforms; it must support real-time scheduling and task isolation across mixed-criticality workloads; and it must enable dynamic execution of precompiled applications without requiring platform-specific recompilation.

Existing operating systems typically fall short on one or more of these requirements. For example, lightweight RTOSes like FreeRTOS offer deterministic scheduling and minimal overhead, but lack support for dynamic task deployment and mixed-criticality scheduling. In contrast, general-purpose real-time operating systems like PREEMPT-RT Linux provide advanced scheduling and deployment capabilities but impose a significant memory and processing footprint unsuitable for many embedded platforms.

This chapter introduces Axcan OS, a custom operating system framework built as an extension to the FreeRTOS microkernel. Axcan OS forms the runtime foundation of the breathable system by enabling cross-platform, real-time execution of position-independent applications. It provides a unified abstraction for both physical ECUs and virtual nodes hosted in mobile edge computing (MEC) environments.

Axcan OS introduces several key features that address the challenges of distributed real-time execution in the proposed system:

- Dynamic task deployment: a lightweight remote deployment manager receives and launches application binaries as decided by the system master node.
- Cross-platform execution: applications are compiled as position-independent ELF files that can be executed unmodified across compatible architectures. Furthermore, system service abstraction provide consistent interfaces for system managed functions across heterogeneous nodes.
- Real-time capabilities: supports various scheduling policies, including preliminary mixed-criticality strategies, and provides global time synchronization using Precision Time Protocol (PTP).

The name of the operating system comes from the Nahuatl word *axcan*, meaning “now” or “immediately,” reflecting the real-time responsiveness envisioned for the system.

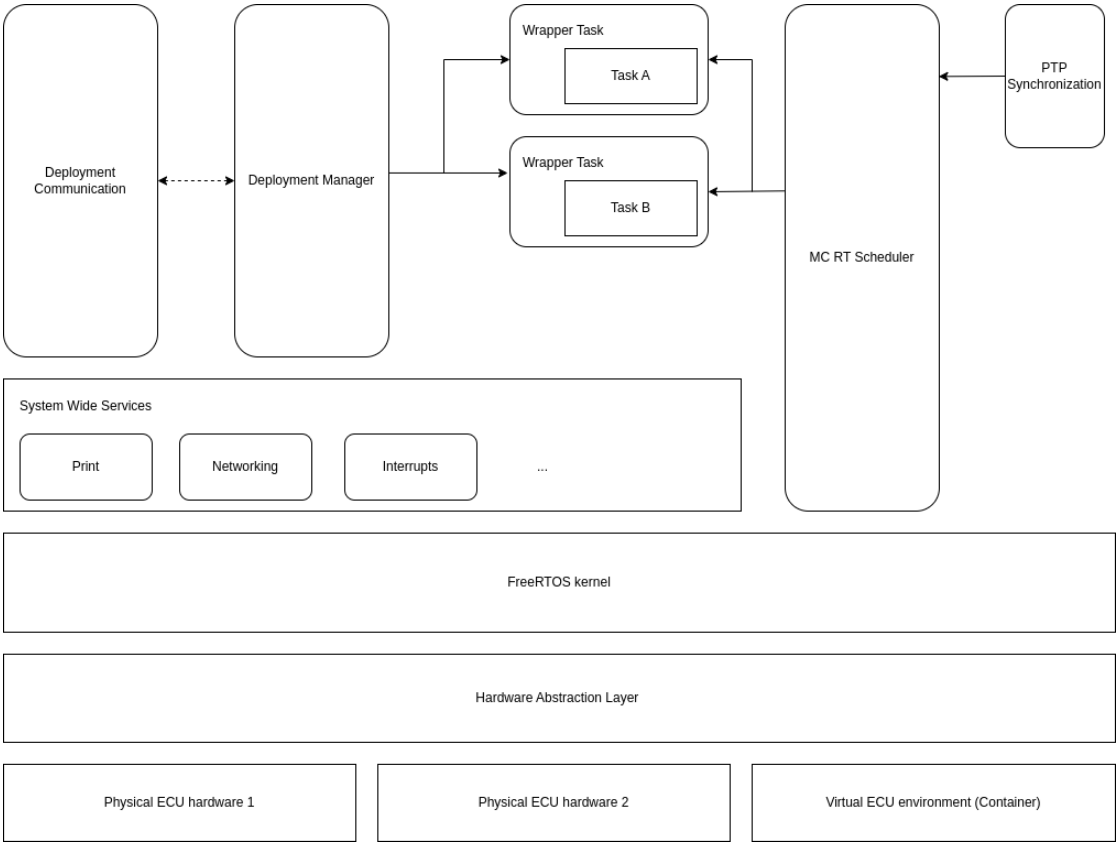

Figure 5.1: Axcan OS Architecture

The remainder of this chapter details the design, implementation, and integration of Axcan OS within the broader architecture, including its current limitations and future development directions.

5.1 Architecture

The architecture of Axcan OS is based on a microkernel approach extending the kernel of FreeRTOS as shown in Fig. 5.1. The diagram illustrates the core components of the system and their interactions. Each module contributes to fulfilling the requirements outlined in the introduction to this chapter:

- FreeRTOS Kernel and Hardware Abstraction Layer: These form the foundational runtime environment, providing deterministic task scheduling and hardware interfacing. Axcan OS builds on the FreeRTOS kernel to retain lightweight real-time execution.
- Task Deployment Communication Handler: Manages communication with the

master node, including the reception of task binaries and instructions, and the transmission of runtime status information such as health metrics or migration readiness.

- Task Deployment Manager: Controls the full task lifecycle, including loading ELF applications into memory, instantiating them, starting or stopping execution, and handling checkpoint data for migration purposes. The deployment manager creates instances of the wrapper task that will contain the actual applications.

- Wrapper Task: Acts as a container for dynamically deployed user applications. It enables runtime execution of position-independent code by handling memory allocation, real-time constraints, and state checkpointing. It also provides user tasks with access to essential system functions through controlled interfaces.

- Extended Scheduler: Selects tasks for executing according to a policy. Configurable for implementing different simple real-time scheduling policies, not only non-criticality-aware ones like rate-monotonic, deadline-monotonic, and earliest-deadline-first, but also mixed-criticality aware approaches.

- System function services: Provides a minimal API for commonly needed services such as network communication or logging. These are abstracted through the microkernel and made available to user applications via the wrapper task, compensating for the lack of shared libraries in the FreeRTOS environment.

- PTP Synchronization: Ensures global time alignment across nodes using the Precision Time Protocol (PTP), which is critical for maintaining timing guarantees and consistency during task migration.

Collectively, these components enable Axcan OS to serve as the runtime backbone for both physical and virtual ECU nodes within the breathable distributed system. The architecture ensures that all nodes — regardless of their underlying hardware or runtime context — can execute precompiled tasks under real-time constraints, coordinate with the master node, and participate in runtime task migration. Furthermore, Axcan OS is envisioned to work on both single-core and multi-core processors, as the multi-core approach treats each core as a sub-node , running copies of Axcan OS in every core in an asymmetric multi-processing (AMP) fashion, as shown in Fig. 5.2, on the example of the ARM Cortex-A53, the processor present by the embedded devices used in the setup of this work. In such cases, core 0 typically acts as the master core and the rest of the cores as slave cores. The architecture of the OS for master and slave cores is almost identical, except for the deployment communication, as inter-core communication and internal task partitioning is also taken into account. The master node is also responsible for the management of system services, as most of them require a dedicated thread to own the resource to avoid conflicts, therefore offering an internal wrapper interface for

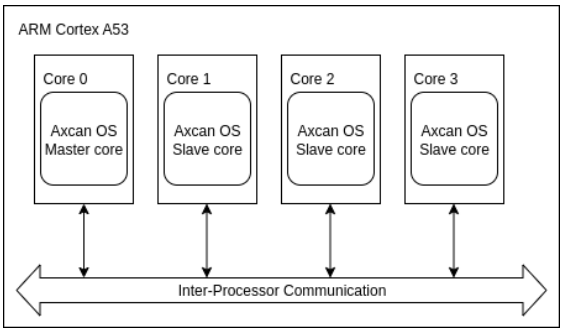

Figure 5.2: Example of a multicore AMP system using Axcan OS

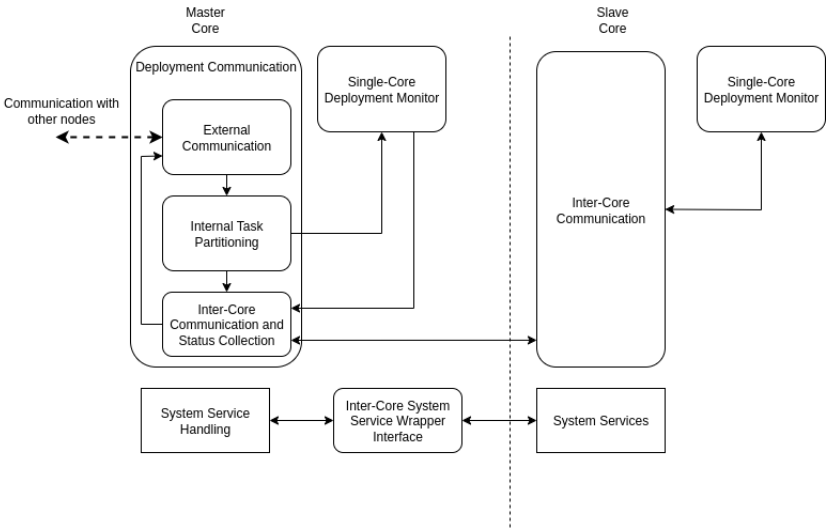

Figure 5.3: OS differences between master and slave cores

system services on the other nodes. These differences are shown in Fig. 5.3, and the internal deployment of tasks is further explained in the next section.

The next sections explain into further detail how Axcan's components interact with each other to implement the features presented in the introduction of this chapter.

5.2 Dynamic Task Deployment

The vision of the breathable system requires ECUs to be able to execute different task loads depending on dynamic needs of the system, which change depending on different factors, such as external events and the availability of hardware resources. As one ECU represents a single node of the distributed system, and the execution of tasks is coordinated by the master node, Axcan includes a pair of tasks that act as an interface for deploying tasks remotely from the master node. First, the deployment communication thread is responsible for establishing a constant communication with the master node, exchanging status information about device health and current task load and status of each task. Also, this thread is responsible for receiving instructions from the master node regarding the execution of the assigned task set, which is then kept in a shared memory block and passed on to the deployment management thread. The deployment management thread is then responsible for handling the startup of the tasks and monitoring their status periodically. The startup includes first receiving the ELF and checkpoint files necessary for task execution, then allocating and initializing the memory required by the task, and finally creating the task thread and registering it in the mixed-criticality scheduler, which is then responsible for controlling when the CPU is given to the task. The monitoring sub-thread is then responsible for periodically checking whether the task is executing correctly, how many jobs have been executed, how many deadlines have been missed, etc. Furthermore, the monitoring is responsible also for pausing / stopping the tasks when told so by the master node, and for preparing and triggering the transfer of task files to another device when migration needs to occur.

5.2.1 Multi-Core Deployment

As shown in 5.3, multi-core nodes behave in a slightly different manner. First, the master core is the only one responsible for communicating with the master node for receiving deployed tasks. After receiving a task set, the master core distributes tasks internally via a greedy partitioning algorithm, aiming for a balanced task load among all cores. This algorithm considers an additional management load for the master core. A further extension of the partitioning is envisioned to handle mixed-criticality and avoiding the clash of multiple-criticalities on independent cores. After the partitioning process has been completed, the master core sends internally the instructions to the slave cores and starts its own task load via the deployment monitor as described above.

Slave cores then replace the communication task for an inter-core communication task, responsible for performing the intermediate communication with the master core, and then perform their normal workflow with the deployment monitor task. In both cases, the monitor task works in exactly the same manner as for single-core systems. The process for sending device and task status remains almost unchanged, only having the master-core collect the status of itself and the slave cores to send a general device status message to the master node in the distributed system.

5.3 Mixed-Criticality Scheduler

The first step in achieving mixed criticality and respecting real-time behavior for the system applications is a scheduler that can achieve both goals. For this purpose, an extension of ESFREE (add reference), a project that implements a few dynamic and static real-time scheduling policies, including Earliest Deadline First (EDF), which for single core is ideally optimal (add references). This project is used as a base as the off-the-shelf FreeRTOS code only offers a priority-based scheduler, without any priority-assignment policies. In this work, the EDF policy is extended to cope with different levels of criticality by implementing the EDF with virtual deadlines for MC, as proposed in (add ref.).

**** Need to finalize implementing scheduling policies, then explain that into detail***

5.4 Cross-Platform Task Loading

To achieve the goal of executing the same application files on both physical and virtual ECU devices, the system integrates an ELF loader. This ELF loader, implemented to a great extent in collaboration with Yinbo Zhou in the scope of his master's thesis, enables the execution of multiple tasks, packed in the form of pre-compiled ELF files and with an additional data file. This approach tackles both challenges of allowing the dynamic loading of tasks and of stateful migration to the extent desired. In order for applications to be compatible with the task loader, they need to fulfill some requirements when compiled. The implemented loader is lightweight and meant to load the same application files on on multiple underlying hardware / software stacks, as shown in Fig. 5.4, as long as an Axcan OS port runs on the target and the CPU architecture is the same as the target architecture for the compiled application, which in our case is always ARM Aarch64.

5.4.1 Hardware Abstraction Layer

In order to achieve full compatibility across different platforms, an abstraction layer is added under Axcan OS. This layer ensures applications can run on any target platform independently of the hardware and software stack, apart from the architecture.

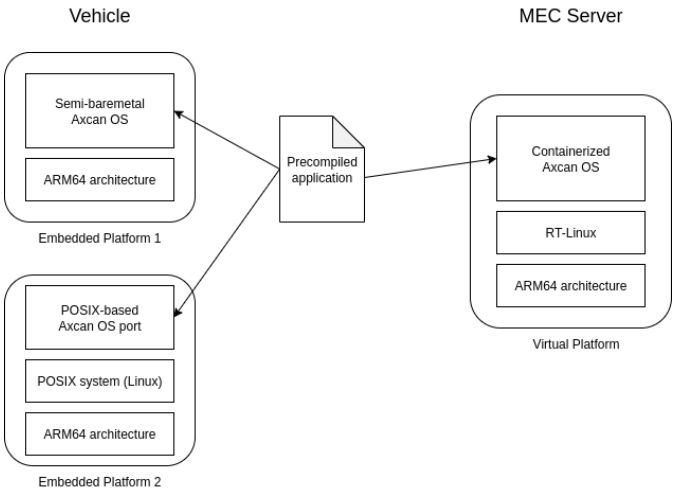

Figure 5.4: Cross-Platform Task Compatibility

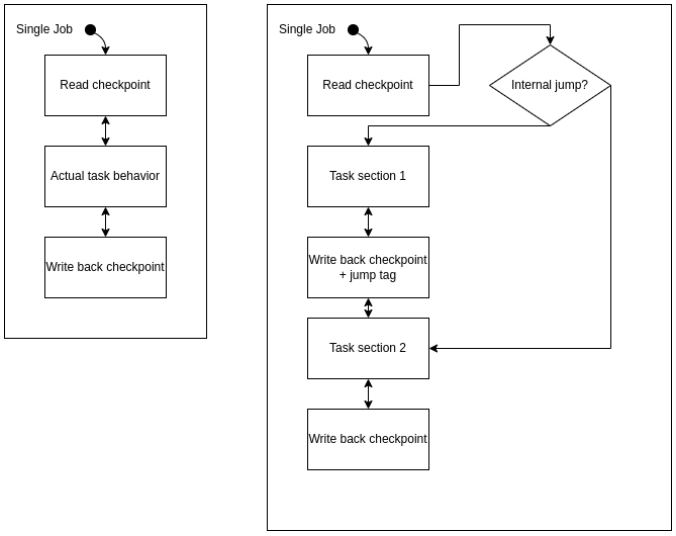

Figure 5.5: User Application Flow

The use of FreeRTOS kernel official ports already allows an abstraction layer in the form of FreeRTOS features, but due to the lightweight implementation of FreeRTOS, some features remain specific to the underlying platform, which is also undesired when aiming to load the same ELF file on different platforms. To achieve this, two core features are included as part of the extended kernel of Axcan OS. First, most of the system calls are encapsulated in wrappers, e.g.: the console output, network features, memory management, access to other peripherals, which are then offered via a standard signature that internally calls the proper underlying code. Second, the ELF loader, apart from processing the ELF and performing the needed relocations, needs to allocate memory for ELF execution, which should be aligned properly for the respective platform. This means that for hardware specific ports of FreeRTOS, which run on a bare metal system, memory can be allocated without any underlying considerations, but also requiring some memory management handling if no memory management unit (MMU) is available on the port. On the other hand, ports based on the POSIX port of FreeRTOS (e.g. the container approach running on a Linux server or even versions running as processes on top of UNIX based OSs) should take into consideration that the underlying OS is responsible for memory management, often not allowing standard *malloc* calls to hold executable code. This specific case means that alternative memory allocation functions, such as *mmap*, which are more flexible but require a more explicit memory handling in code, should be used.

5.4.2 User Application Requirements

User applications need to meet two requirements to be compatible with Axcan OS. First, they need to be compiled with the right flags, particularly by enabling position independent execution (PIE) and to avoid integration of standard C libraries and system startup functions (they should use newlib to allow for applications to be encapsulated without dynamic library linking). Second, they should use the Axcan OS header and library files, which will provide access to basic system functions and core features, such as the checkpoint mechanism, the resource table for checkpoint variable handling and the proper entrypoint handling. System services are offered through global function pointers. All those elements are initialized by the operating system and passed to user applications through a data structure in shared memory.

Furthermore, by design user applications are periodic and must define and keep track of the state-relevant variables for their further execution, meaning that the main function acts as a single job execution, and any execution context needed for a job must be loaded at the beginning of the main and stored back at the end. This represents the minimal checkpoint version where state is only resumed from the beginning of a job. Further checkpoints can also be added in the middle of the code in the form of internal jumps that need to be executed at the start of the main function, allowing the job to resume also from internal points. These two cases are shown in Fig. 5.5.

5.5 Summary

This chapter introduced Axcan OS, the lightweight real-time operating system designed to serve as the execution environment for both physical and virtual nodes in the proposed breathable distributed system. Building upon the FreeRTOS microkernel, Axcan OS was extended with custom components to meet the core functional requirements for allowing single nodes to be integrated in the overall distributed real-time system, namely dynamic deployment of real-time tasks, task migration and execution across heterogeneous platforms and handling of tasks with different criticalities.

The modular architecture of Axcan OS, along with its internal mechanisms, as explained in this chapter, implement the backbone for the behavior of the ECU nodes, making it possible for them to work as parts of a single system. This integration of different nodes is the base for the features explained in the next chapters, such as the task migration strategies, the real-time considerations in the distributed system and the task allocation to specific nodes.

It is relevant to note that at the moment of writing, Axcan OS is implemented as a proof-of-concept framework. Further work is needed to provide a wider set of system services to user applications, and possibly to provide more efficient communication mechanisms among nodes in the distributed system.

6 Task Migration Implementation

6.1 Statefulness and Checkpoint Mechanism

In order to keep the running state of the system tasks during migration (i.e. making it stateful), a checkpoint mechanism should be implemented. In this context, checkpoint refers to a data file containing all relevant variables to store or reload a snapshot of the task's state, whereas mechanism refers to the whole process of storing and reloading the task. There is a trade-off between the precision of the checkpoint mechanism and its simplicity and amount of checkpoint data. For example, a task's state could be very precisely restored down to a specific instruction if stuff like program counter and full task context (e.g. stack and heap, and nested data structure references) can be snapshotted. Another advantage of this strategy is that tasks could be implemented without being specific to the system, but the drawback is that it might require keeping a lot of data and a more complicated implementation at kernel level. On the other hand, checkpoints can be defined in the code at some strategic steps, along with the most relevant processed variables (e.g. for an object detection algorithm after every processing step of an image). This might mean that some work could be lost after migration, but the checkpoint structure would be simple and the mechanism would involve only some code jumps, although the task would need to be specifically implemented to work with such a mechanism. The approach for the checkpoint mechanism in this work is closer to the latter, and the reasoning behind this design decision, as well as the details of its implementation are described next.

First of all, let us recall the nature of the system and the executed tasks. As mentioned before, these tasks are all assumed to be periodic, in each iteration (job) needing to process some information, possibly based on sensors or other peripherals, and generating an output, that might also depend on previous iterations. Given this behavior, it is assumed that at least, a checkpoint can be stored after a full iteration while keeping a meaningful state. Additionally, other internal checkpoints could be made by keeping jump tags if higher level of granularity is needed. During development phase, the checkpoint has to be designed in the form of a task structure which can contain all the relevant data and be stored in a file. At runtime, task reloading requires three fundamental steps, as shown in diagram...: 1) reading the existing checkpoint to get previous state, 2) performing the actual job, and 3) writing back the new state in a checkpoint. Additionally, if jump tags are stored, after retrieving the previous state, the corresponding jump is performed, while it is also important to save the checkpoint before the next sub step of the job (i.e. next jump tag), including a hint to the jump

tag. The design decision to follow this strategy was taken considering the smaller overhead at runtime and during implementation of the OS. Also, since aforementioned steps 1 and 3 (reading and writing back the state) would be the same for most simple applications, they can be pre-implemented via a library, making it only necessary to define the task state data structure, and otherwise keeping the application code fairly simple. Furthermore, flexibility for fine granularity in the checkpoint keeping is possible through the mentioned jump tags and by allowing to override the specific behavior of steps 1 and 3.

An example of the above mentioned behavior could be a SLAM (simultaneous localization and mapping) algorithm, which builds a cumulative map of the surroundings based on the received sensor data and finds the current position of the exploring agent (e.g. a vehicle) in that map. In this case, results of previous jobs (the generated map and the agent's position) are also used to generate the results of the current iteration. For this task, a minimal state could be stored by keeping track of those results, but as it is a more complex task with several steps, jump tags could be kept for example after each of the relevant steps: pre-processing sensor data, sensor fusion, identifying agent's position, filling the map with new data. This second behavior allows for a finer granularity for migrating the task while still avoiding the overhead of a full kernel-space snapshot.

While the previous paragraphs define the general behavior of the checkpoint mechanism, it is also necessary to define the way in which it is transmitted from the device currently executing the task (source) to the device that will execute next (target). As mentioned before, the user should define a data structure with all relevant data inside the task code, this will be then accessible through a void pointer from the OS in binary format, which would be further processed to share the checkpoint with the migration target. In general, defining the data structure in this way allows for a variety of implementations, as long as the data stored in it keeps its integrity, and the specific transmission approach will depend on the specific migration strategy according to the criticality of the task and the type of target device as described in the next sections.

6.2 High Criticality Strategy

*** MISSING details now, but idea is explained in approach***

- The idea is based on a premise that a shared memory approach would allow faster and less data-intensive transmission for performing the migration

- The task has 3 possible states on each device: off, running and (hot) standby. For tasks that do not require redundancy at runtime, only one (active) device (node in the distributed system) keeps them in running state, while a selection of backup nodes keep it in standby mode. To enable runtime redundancy, multiple nodes

can be active. Nodes that are neither active nor selected as backup keep task in an off state.

- Both running and hot-standby modes keep the task loaded on the system and ready to execute, but standby mode does not make task perform any work, while running mode does.
- State is kept through a shared memory approach. In the scope of this work (so far), the concept is explored using multiple cores in a single multicore processor as if they were independent nodes, with the processor memory working as the shared memory.
- Conceptually, this memory can represent a distributed shared memory (DSM), which could be further integrated using a real-time communication protocol to keep time-bound behavior. Example of this approach might be... (Do proper referencing...) https://doi-org.eaccess.tum.edu/10.1007/s10586-017-1140-9 https://dl.acm.org/doi/abs/10.1007/s10586-017-1140-9
- The idea would be to keep a fixed reference to this space in memory for all running and standby instances, which would be read before and updated after each executed job.

6.3 Low Criticality Strategy

- Based greatly on Yinbo's thesis
- Here, idea is that tasks can be precompiled to ELF files which can be loaded dynamically upon request
- Task data are stored in a separate file that can be loaded with the ELF and later stored upon request
- Explain how tasks must be compiled and prepared to provide valid ELF files.
- Loader would work with any scheduling algorithm, ensuring that MC can be respected in the system

6.4 MEC: Virtual Devices

- Based greatly on Roberto's thesis
- Here, idea is that the MEC can host containerized versions of the devices – Implementation specific details for the containers are out of scope in the thesis, as they are part of Bernhard's idea

- Explain briefly that it is assumed the implementation of the containerized version of the tasks is fully implemented in the scope of future work
- Explain how the migration to and from the cloud works
- Explain how the migration to and from the MEC is integrated into the system architecture, especially how it is considered when taking a decision to migrate to MEC or to another physical device
- Discuss missing implementation details

The work presented in (cite ROberto's thesis) showed the possibility of containerizing a POSIX port of FreeRTOS and running it on a Linux server. Following this concept, Axcan OS was ported to make it compatible with such a container approach, making it possible to load on the virtual devices exactly the same ELF executables that are loaded for the real embedded devices, avoiding the need for recompiling the tasks. Furthermore, the features of Axcan OS allow the tasks to access the system services in a seamless manner.

The container-compatible POSIX port of Axcan OS requires a different management of memory and peripherals, especially due to the restrictions introduced by the Linux kernel, for example due to the exception level or the rights assigned to memory allocated with functions like malloc or mmap. However, Axcan OS offers an abstraction layer that offers the features to the user applications on the same way on both platforms.

7 System Integration

7.1 Real-Time Constraints in the Distributed System

7.2 Adaption of the ML algorithm

7.3 Virtual and Physical Devices

8 Results and Evaluation

8.1 Test Setup

8.2 Discussion

9 Conclusion & Outlook

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Bibliography

[Bar+11] S. K. Baruah, V. Bonifaci, G. D'Angelo, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie. "Mixed-Criticality Scheduling of Sporadic Task Systems." In: *Algorithms – ESA 2011*. Ed. by C. Demetrescu and M. M. Halldórsson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 555–566. ISBN: 978-3-642-23719-5.

[Bau+18] M. Baunach, R. Martins Gomes, M. Malenko, F. Mauroner, L. Batista Ribeiro, and T. Scheipel. "Smart mobility of the future - a challenge for embedded automotive systems." In: *e & i Elektrotechnik und Informationstechnik*. Vol. 135. 4. 2018, pp. 304–308. DOI: 10.1007/s00502-018-0623-6.

[BBD11] S. Baruah, A. Burns, and R. Davis. "Response-Time Analysis for Mixed Criticality Systems." In: *2011 IEEE 32nd Real-Time Systems Symposium*. 2011, pp. 34–43. DOI: 10.1109/RTSS.2011.12.

[Bli19] B. Blieninger. *Online Machine Learning for improved decision making regarding the migration of automotive software components at runtime*. 2019.

[Bur+18] O. Burkacky, J. Deichmann, G. Doll, and C. Knochenauer. *Rethinking Car Software and Electronics Architecture*. Tech. rep. McKinsey & Company, Feb. 2018.

[But11] G. C. Buttazzo. *Hard Real-Time Computing Systems*. 2011. DOI: 10.1007/978-1-4614-0676-1.

[Del21] O. I. Delgadillo Ruiz. *Hardware-in-the-Loop Test Setup for a Generalistic Approach to Machine-Learning-Based Schedulability Analysis*. 2021.

[EKB15] S. Eckl, D. Krefft, and U. Baumgarten. "KIA4SM - Cooperative Integration Architecture for Future Smart Mobility Solutions." In: *Conference on Future Automotive Technology (CoFAT)*. Vol. 4. Apr. 2015.

[Fle13] T. D. Fleming. *Extending Mixed Criticality Scheduling*. Tech. rep. The University of York, 2013.

[GRP15] R. Gratia, T. Robert, and L. Pautet. "Scheduling of mixed-criticality systems with RUN." In: *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*. 2015, pp. 1–8. DOI: 10.1109/ETFA.2015.7301484.

[HC16] C. Hainz and A. Chauhan. *Automotive Change Drivers for the Next Decade*. Tech. rep. EY Global Automotive & Transportation Sector, 2016.

[KSK18] F. Kuhnert, C. Stürmer, and A. Koster. *Five Trends transforming the Automotive Industry*. Tech. rep. PricewaterhouseCoopers GmbH Wirtschaftsprüfungsgesellschaft, 2018.

[LB12] H. Li and S. Baruah. "Outstanding Paper Award: Global Mixed-Criticality Scheduling on Multiprocessors." In: *2012 24th Euromicro Conference on Real-Time Systems*. 2012, pp. 166–175. DOI: 10.1109/ECRTS.2012.41.

[Li+13] L. Li, R. Li, L. Huang, R. Wu, and L. Zeng. "A New RTA Based Scheduling Algorithm for Mixed-Criticality Systems." In: *2013 IEEE 16th International Conference on Computational Science and Engineering*. 2013, pp. 722–729. DOI: 10.1109/CSE.2013.111.

[Mun+17] P. Mundhenk, G. Tibba, L. Zhang, F. Reimann, D. Roy, and S. Chakraborty. "Dynamic platforms for uncertainty management in future automotive E/E architectures." In: *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2017, pp. 1–6. DOI: 10.1145/3061639.3072950.

[RE18] S. Ramanathan and A. Easwaran. "Mixed-Criticality Scheduling on Multiprocessors with Service Guarantees." In: *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*. 2018, pp. 17–24. DOI: 10.1109/ISORC.2018.00011.

[Tai19] S. Taieb. *Deep Learning Based Schedulability Analysis for Migration of Software Components at Runtime*. 2019.

[Utz19] T. Utz. *Vergleich von heuristischen und Deep Learning basierten Schedulability Analyse Verfahren für die Migration von Softwarekomponenten zur Laufzeit*. 2019.

[Vip+14] K. Vipin, S. Shreejith, S. A. Fahmy, and A. Easwaran. "Mapping Time-Critical Safety-Critical Cyber Physical Systems to Hybrid FPGAs." In: *2014 IEEE International Conference on Cyber-Physical Systems, Networks, and Applications*. 2014, pp. 31–36. DOI: 10.1109/CPSNA.2014.14.

[Vip18] K. Vipin. "CANNoC: An open-source NoC architecture for ECU consolidation." In: *2018 IEEE 61st International Midwest Symposium on Circuits and Systems (MWSCAS)*. 2018, pp. 940–943. DOI: 10.1109/MWSCAS.2018.8624006.

[ZLL19] L. Zeng, Y. Lei, and Y. Li. "A Semi-Partition Algorithm for Mixed-Criticality Tasks in Multiprocessor Platform." In: *2019 IEEE 10th International Conference on Software Engineering and Service Science (ICSESS)*. 2019, pp. 694–697. DOI: 10.1109/ICSESS47205.2019.9040792.

[ZWL17] X. Zhao, Y. Wei, and W. Li. "The Improved Earliest Deadline First with Virtual Deadlines Mixed-Criticality Scheduling Algorithm." In: *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)*. 2017, pp. 444–448. DOI: 10.1109/ISPA/IUCC.2017.00072.

A Additional Stuff

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.