TECHNISCHE UNIVERSITÄT MÜNCHEN
INSTITUT FÜR INFORMATIK

Energy-aware Computing in Embedded Systems and its Operating System Support

Andreas Barthels

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

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Johann Schlichter
Prüfer der Dissertation:	
	1. Univ.-Prof. Dr. Uwe Baumgarten
	2. Univ.-Prof. Dr. Andreas Herkersdorf

Die Dissertation wurde am 08.04.2014 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 20.08.2014 angenommen.

Abstract

The thesis presents a novel approach to establish energy-aware computing in embedded systems in an early design phase. The approach supports both explicit and implicit mechanisms. Embedded systems underlie strict criteria for functional safety, thus the design and software integration process are very intensive. The approach and techniques are formally defined and evaluated in a test bench.

The model case for the evaluation stems from the automotive domain. Modern vehicles serve as a model class of complex distributed systems with special requirements. Because of the limited capacity and dynamics of energy storage and conversion systems, the elaborated mechanisms are designed with scheduling of combined software and power in mind.

In order to be able to meet safety requirements, the modeling and the design of the mechanisms is conducted in an offline manner. This allows to formally verify the scheduling schemes in respect to voltage stability effects in the automotive power net.

The meta-model of the underlying concept is crafted with focus on implementation within current embedded systems in mind. As a reference, a corresponding scheduling paradigm was integrated into the Linux operating system kernel and tested on a modern embedded system platform.

Kurzfassung

Die Arbeit präsentiert einen neuartigen Ansatz um energiebewusste Datenverarbeitung im Betriebssystem von eingebetteten Systemen zu ermöglichen. Der Ansatz unterstützt sowohl implizite, als auch explizite Mechanismen. Eingebettete Systeme unterliegen strikten Kriterien für funktionale Sicherheit, weswegen der Entwurfs- und Integrationsprozess der verwendeten Software sehr intensiv ist. Das Konzept wird formal definiert und in einem Prüfstand evaluiert.

Der Ansatz wurde speziell für Automobilsysteme entwickelt, die als Beispiel für komplexe verteilte Systeme mit besonderen Anforderungen dienen. Da im Automobilbereich die Kapazität und Dynamik der Energiespeicher und -wandlungssysteme begrenzt sind, wurde ein Ansatz gewählt, der bewusst Software sowie Leistungsmodi und Zeitscheiben im System koordiniert. Kerngedanke ist die Abstraktion und Ausführung gegebener Leistungspläne im Betriebssystem.

Um die Sicherheitsanforderungen erfüllen zu können, wird davon ausgegangen, dass die Modellierung und der Systementwurf offline durchgeführt werden. Hierdurch lässt sich die zeitliche Ablaufplanung im Hinblick auf Spannungsstabilität im Energiebordnetz auslegen.

Das Metamodell der zugrundeliegenden Konzepte berücksichtigt insbesondere die Implementierbarkeit in eingebetteten Systemen. Zu Demonstrationszwecken wurde das vorgestellte Planungsparadigma in den Linux Betriebssystemkern integriert und auf einer aktuellen Plattform getestet.

Acknowledgements

First and foremost I want to thank my adviser Prof. Dr. Uwe Baumgarten for supporting me throughout my time as a researcher at the Technische Universität München (TUM). He has given me the chance to write this thesis while working on a collaboration project together with the BMW Group and the institutes for integrated systems and energy conversion technology.

I want to thank my colleagues at the mobile distributed systems group, as well as the colleagues at the collaborating institutes. Special thanks go out to the heads of these institutes, Prof. Dr. sc.techn. Andreas Herkersdorf and Prof. Dr.-Ing. Hans-Georg Herzog, for providing me the opportunity to work with them on tools for modeling and simulation, as well as for providing the test bench as part of which the evaluation of this thesis was conducted.

During my time at the TUM, I advised several theses and interdisciplinary projects. The fruitful discussions with colleagues and students, as well as with my research assistants Michael Alberter and Alexandra Buzila, have brought me forward both personally as within our disciplines.

Contents

Part I.

Foundations & Theory

1. Introduction

One of the major topics of the century is environment and resource awareness. This is resembled in a strive for increasing efficiency in economy. Computing has become an integral part of economy and society. Using computers to improve efficiency of complex systems (computing for green), as well as making the computing processes themselves more efficient (green computing) are major goals.
Due to the miniaturization of computing systems, there is a trend to embed them into numerous devices. Embedded systems are found in a multitude of smart devices affecting all sectors of economy, individuals' lifes and scientific experiments. Embedded systems are widely used such as in industrial control systems, household appliances, gadgets, and as personal assistance in the services sector. Last but not least, scientific studies rely on embedded computing systems for gathering data.

In the past decades, embedded systems were integrated into all sorts of transportation systems. Of special complexity are both avionic as well as automotive systems. The automotive domain serves as a model case within this thesis. This domain is rich in interconnected embedded systems.
An important issue in this domain is safe and stable operation. Automotive systems can become unstable and subsystems can be reset by so-called undervoltage lockouts. The voltage levels inherently depend on the sources and sinks for electrical energy as well as their dynamic behavior.
The sources and sinks of electrical energy can be subsumed as energy conversion components connected by a wiring harness, as depicted in BMW Technology Guide (2014). In the past, stability issues were largely tackled by adding larger lead-acid batteries, or by increasing the cross sections of the wires. Because batteries and wires have reached large sizes, the power consumption dynamics of the sinks have increasingly to be taken into focus for stable operation.

Hybrid and electric vehicles additionally have multiple voltage levels for different types of subsystems. Legacy systems work using 14 V power nets. High power electrified systems such as propulsion, steering and stability control are supplied using a higher voltage level. Together with these heterogeneous supply voltage systems, the IT network of the embedded systems is heterogeneous too.

Typically, embedded systems are low-cost and highly specialized to the task at hand. Thus, the operating systems running on these devices have to be lean in order to minimize overhead. From a standpoint of system development, system functions are provided by applications. These applications commonly also take care of power management nowadays. The applications form distributed systems which are assumed to be described using formal models. The models resemble the interaction patterns as well as the inter-dependencies of software units.
The embedded hardware platforms are becoming increasingly powerful and at the same time more versatile. Thus, new and flexible concepts are needed in order to integrate

and (re-)combine different pieces of software on a platform.

1.1. Foundations

There is a wide range of previous and related work for energy aware computing. In this thesis, energy awareness is seeked by working on instantaneous power consumption over time. The consumption stems from hardware converting in between forms of energy. Energy aware computing started with defining different modes of operation for electronic components. These modes in general can be activated by special electronic circuitry which enables electronic systems to power manage themselves.

1.1.1. Power Management in Hardware

Common operating modes of computing systems include shutting down and halting the system, or running at a certain processing speed.

Before Advanced Power Management (APM) (Microsoft Corp., 1996) and Advanced Configuration and Power Interface (ACPI) (ACPI, 2013) were introduced, the changing of operating modes or power states had to be triggered manually by the user. In the x86 architecture, until 80486, computers featured a Turbo-button and had to be turned off manually once the system was halted.
Both APM and ACPI define a set of operating modes in between full on and off, allowing different stand-by operations in between. In general it can be said that the more is turned off, the more costful in time and energy it is to turn the machine to full on again.

Because efficiency of computers have long been researched, hardware manufacturers have incorporated an increasing number of modes like different power and frequency domains for different parts of the hardware. There are a multitude of on- and offline methods for utilizing these already (Benini et al., 2000; Irani et al., 2003; Jejurikar and Gupta, 2004).

As a recent contribution, Moreno and de Niz (2012) have focused on voltage and frequency scaling for uniform multiprocessors. They present an algorithm called Growing Minimum Frequency and focus on maximizing energy efficiency. Next to being able to dynamically switch modes of computing hardware, these modes need typically to be triggered by software over time.

1.1.2. Embedded Systems Scheduling

Planning resource assignment over time is called scheduling. Scheduling is done by operating systems and has seen many different variants.
Since embedded systems are often used for controlling physical processes, tasks mostly have requirements on real-time execution. On the one hand, tasks may have requirements on soft real-time, meaning they need a specific amount of bandwidth over time. They must not necessarily be run at a very specific time point, and may well be interrupted as long as the bandwidth is reached in the end. Hard real-time tasks need a resource at a given time and need to finish by some deadline.

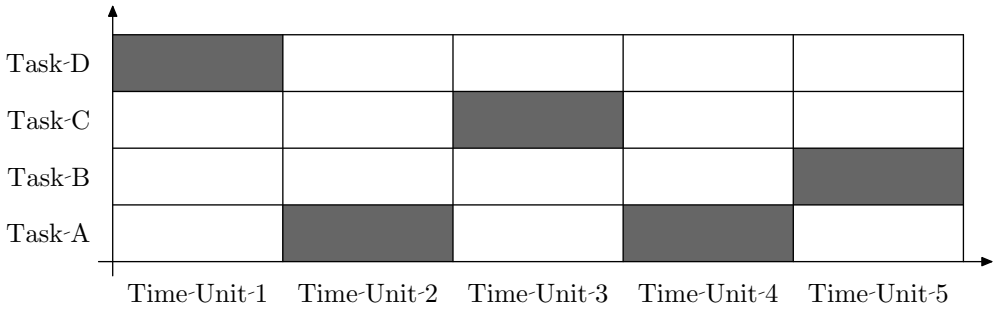


Figure 1.1.: Periodic Scheduling with Aligned Periods

Periodic Scheduling

Periodic scheduling is used to support so-called hard real-time tasks. It is a purely time triggered scheme which has long been under investigation. Every task has a time period and a time offset which define the activation. Using this scheme, and combining it with worst-case execution times of tasks, schedules can be well created and understood offline. Using worst-case times, one can effectively preallocate execution windows for each task. Under normal operation of the software units, the schedule is well predictable despite of the actual execution time within each window.

This periodic form of resource assignment over time is important when system properties need to be guaranteed. Since the time windows meet the worst case system behavior, the resource utilization is typically significantly lower than 100 %. Thus, this paradigm typically allows for energy savings during idle phases, but it must be ensured that the system predictability is maintained.

Aperiodic Scheduling

Aperiodic scheduling is less predictable in general. Both hard as well as soft real-time tasks can be aperiodic. Tasks are generally triggered by events, which can be timer expirations, or system inputs.

The nature of aperiodic systems is in general hardly predictable. Typically, tasks are assumed to be triggered according to stochastic processes. The superposition of task activations can lead to many different schedules over time. To resolve conflicts of tasks requesting the same resource at the same time, prioritization and preemption can be used.

Figure 1.2 shows an example in which a higher prioritized task arrives during the runtime of a lower prioritized task. In a system without preemption, the next task is selected based upon priority once the current task finishes. Additionally, a system may support preemption, interrupting the lower prioritized task and resuming its operation after the higher prioritized task has finished.

1.1.3. Power Management and Interaction Paradigms

Operating systems as well as middlewares incorporate interfaces in between distributed applications, allowing a multitude of interaction schemes. The interfaces may well be

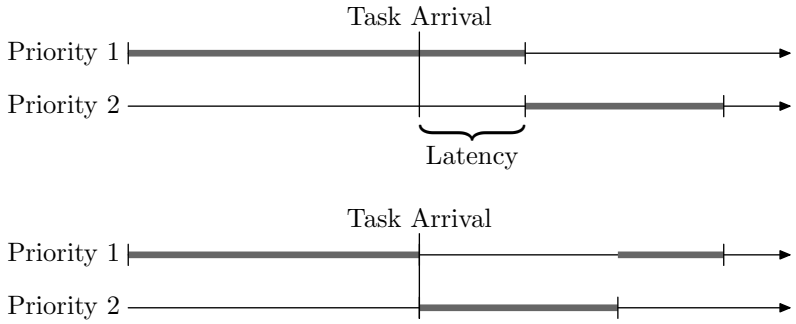


Figure 1.2.: Scheduling Both Without and With Preemption

used to coordinate power management mechanisms as part of the application, as well as incorporating a management mechanism in themselves.

Eugster et al. (2003) characterize and compare common software design patterns. They distinguish the following patterns:

Message Passing The simplest form of interaction scheme is passing of messages as plain datagrams. Distributed systems relying on message passing can best be described as synchronous data flow graphs. As such senders and receivers are coupled in time, space, and synchronization.

In the scope of software running on top of an operating system, different services can be utilized, such as name resolution, routing, ordered delivery and queuing.

Message Queuing Queuing is a common mechanism within operating systems, provided on multiple levels, e.g., by the device drivers, as well as sockets or the Transmission Control Protocol (TCP). Using queuing, senders and receivers can be decoupled in synchronization, allowing receivers to read asynchronously.

Remote Procedure Calls Remote Procedure Calls (RPCs) use message passing to semantically interact among software components. A software component may provide interfaces for interaction with other components. For RPC, a mediating layer between components is needed. This layer can provide RPC synchronously as well as asynchronously, which leads to the notion of notifications.

Notifications Notifications asynchronously propagate events in the system. Using RPC, notifications may be delivered by invoking specified callback procedures. Notifications are widely used to implement the observer design pattern. Using asynchronous notifications, communicating endpoints are still coupled in time and space, meaning they have to be online at the same time and need a way of communicating with each other without an intermediary component. The coupling in time and space can be lifted by adding an intermediate component such as a shared space.

Shared Spaces Shared spaces form an intermediate entity between endpoints. Endpoints are decoupled in time and space; they do not address each other directly and do not need to be up and running at the same time. The shared space is filled

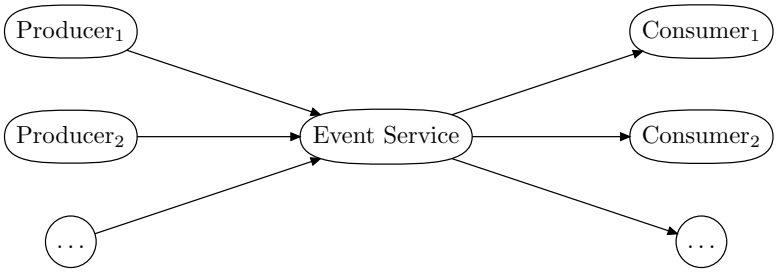


Figure 1.3.: Complex Interaction Pattern for Decoupling of Endpoints

asynchronously and read synchronously. Providing additional decoupling from synchronization results in the implementation of a publish/subscribe scheme,

Publish/Subscribe Publish/subscribe is a means to decouple entities from time, space, and additionally from synchronization. This pattern is similar to the shared space, but subscribers are asynchronously notified of data being published by publishers. This scheme provides the most sophisticated means of decoupling interacting components and thus allowing for great flexibility in system integration and operation.

These interaction paradigms are subsumed in middlewares. As part of the software engineering process, the description of participating components enable system-wide adoption and adjustments of Quality of Service (QoS) as described by Schantz et al. (2003).

1.1.4. Task Graphs

Since there is no standardized set of tasks and interaction patterns, tools for generating sets of tasks were developed. One such tool is Task-Graphs-for-Free (TGFF, Dick et al. (1998)). It randomly generates sets of independent graphs as well as dependent directed acyclic graphs of tasks. These graphs define an ordering, within which the data flow and thus execution must occur. Execution properties, like worst-case execution time as well as end-to-end deadlines and communication resources can be included.

The task graphs were initially used for synchronous aperiodic scheduling. If some deadlines are omitted or if multiple graphs are executed on the same system, it may well serve for generating asynchronous patterns.

Synchronous Data Flow Graphs (SDF)

Synchronous Data Flow Graphs feature cyclic input dependencies and thus are only applicable to synchronous periodic scheduling (Stuijk et al., 2006). Due to the strong coupling in time, space and synchronization, they are mostly used in integrated circuit design, e.g. for accelerating multimedia stream processing.
Thus, they allow for analysis of multicore, highly parallel multiprocessing applications. They are widely used for deadlock and consistency analysis. Consistency means all outputs of a node are being processed and the memory needed for execution is bounded.

Different types of task graphs may serve as a model for the logical interaction pattern of software components, which is one aspect of modeling a distributed system such as a vehicle.

1.1.5. Modeling

In order to tackle the increasing complexity, model-driven software development became the basis for interconnected embedded systems (Karsai et al., 2003).

The modeling distinguishes the following layers:

Requirements Level when a function is defined, the requirements are specified first and foremost. These requirements comprise the behavior and functionality of the system. An implementation of these requirements will then touch the following, more specific levels.

Logical Level describes the software from a logical point of view. Within this level, no implementation specific knowledge is modeled. Software is split up into logical units along with their communication dependencies. The logic has to fulfill the respective requirements. Using multi purpose hardware, these logical units can be deployed and implemented in various ways on the technical level.

Technical Level contains the technical architecture of the system; the kinds of and numbers of embedded systems. Which is, more specifically, a description of the computing hardware and peripherals, as well as communication links. Each hardware component of such an embedded system has a set of features, and different operating modes.

Implementation Level handles the implementation of the abstract logical functions to specific hardware. In here, the software is again divided into units (i.e. components), which do not have to be the same as in the logical level. Each component can be tested to meet the requirements and is often monitored during runtime to ensure correct behavior on the concrete hardware platform, which comes into play within the integration level.

Integration Level ships implemented software units onto hardware units and provides scheduling and resource assignments. On the integration level, all abstract models are subsumed. This level is needed to deduce physical properties of the system, such as timing and dynamic behavior. Integrating software components onto specific hardware is done by configuring the operating system so as to meet all requirements. These requirements are typically according to static or dynamic resource allocation. Static allocation is the fixed reservation of a resource during runtime. Dynamic allocation is referring to scheduling resource usage over time.

Figure 1.4 depicts an example for combining a logical and technical model with an integration view. The jobs and communication patterns stem from the logical layer, the hardware subsystems build the technical layer. Both are combined in the integration view by statically partitioning jobs to hardware components.

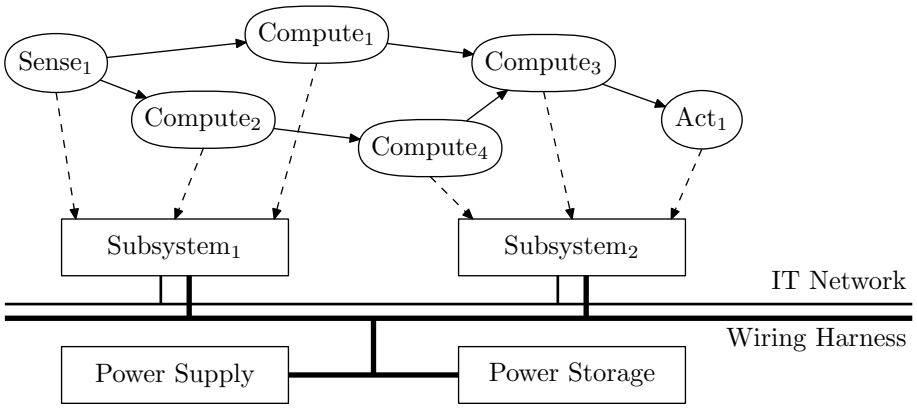


Figure 1.4.: Illustration of jobs interconnected by communication patterns and distributed onto embedded subsystems. Illustration taken from Barthels et al. (2011).

Unified Modeling Language (UML)

In the past decades, the UML was elaborated by a multitude of contributors. The language copes with all levels of modeling by, e.g., specifying requirements, such as actor, sequence, and state diagrams, as well as logical models such as class diagrams. For specific modeling domains, such as automotive systems with their special hardware equipment, special tools aiding in system development have been developed (Haberl, 2011). The UML was extended to the Systems Modeling Language (SysML), which leads to computer-aided system design.

Computer-Aided System Engineering (CASE)

Modeling is central to CASE-tools. The tools handle different aspects of the levels mentioned before. PREEvision by Vector Informatik GmbH (2013), e.g., copes with the requirements, logical, and technical levels. In recent versions, additional support for the implementation and integration levels was added. Depending on the target platform, specialized tools are used for the implementation and integration of components. In the early phase, disregarding actual implementation, the system can already be formally analyzed. For system analysis, so-called model checking methods are being researched. These methods require the transformation of the models into descriptions which are amenable to analysis. These models as well as their algorithmic simulation and analysis are implemented in widely adopted tool-chains, like UPPAAL (Behrmann et al., 2006). These tools can be used to model the processes in distributed embedded systems and to analyze them in terms of model checking.

Model Checking

Common models are Timed Petri Nets (Ramchandani, 1974), and timed automata (Alur and Dill, 1994). These models can be converted into one another, as analyzed by Balaguer et al. (2012). The abstractions and tools for model checking are used for testing

the feasibility of a system design, like a control or resource allocation strategy. They are not designed for being run in the embedded systems as they are.
For modeling control systems and running them as they are, a theory of hybrid automata and abstractions for hybrid systems were developed (Tiwari, 2008). Hybrid systems grasp a mixture of continuous (physical) and discrete (IT) event-based distributed systems. Hybrid automata can be rolled out and integrated into embedded systems to actually perform the functions. They can be checked, or validated beforehand, but they are not as amenable to automated exploration of the design space, due to their expressiveness.

Design Space Exploration

Using tools for model checking, engineers naturally perform the task of design space exploration; a sensitivity analysis regarding performance of different design aspects.
This can be achieved by repetitively tweaking the models and perform checks and simulation on them. The simulation can yield artificial performance measures, which again can be used to rank the solution candidates. Automating this task of exploration, and combining it with optimization was done for the task of mapping software to hardware, by Katoen et al. (2013).
The automation regards two important steps.

1. It is important, that solutions which are analyzed are correct. Thus an enumeration mechanism is needed which inherently produces consistent models. This can be tackled by utilizing solvers for constraint satisfaction problems, such as the one used by Katoen et al. in Microsoft FORMULA (2012).

2. The model has to be simulatable, be it through modeling as a stochastic process (Katoen et al., 2013), or as an accurate simulation as explained by Šimunić et al. (1999), and Walla et al. (2012c).

Once a valid configuration and integration of all components is found, the system can be shipped and runs on an operating system. This operating system abstracts and manages the resources employed by the distributed applications.

Testing-in-the-Loop

The development of distributed systems is often split up using a divide and conquer approach. Each subsystem model or implementation can be tested against its specification. In an early stage of the design process, only models of the subsystem exist. These are tested using the Model-in-the-Loop (MiL) approach. Using this approach, the models are deployed into a test bench and the subsystem dynamics can be analyzed.

If the models successfully pass the tests, they are being implemented and the final subsystem candidate can be tested using Hardware-in-the-Loop (HiL). The concepts presented in this thesis are designed with support for these testing processes in mind.

1.2. Problem Statement

The key challenge is to enable energy efficiency while maintaining safety and enabling further computing-based innovations.

Nowadays, operating systems for real-time systems hardly support dynamic power saving mechanisms. A novel power management paradigm is needed allowing to achieve energy savings as well as supporting transient reactions to system hazards. For increased flexibility and power savings, it is desirable that software units can be redeployed onto different hardware units.

For this to be feasible, suiting abstractions have to be found and services in the operating systems have to be implemented, which support a variety of hardware architectures.

1.3. Related Work

The field of energy aware computing has seen a long history of related work. The summary given in this chapter focuses on aspects of power management in operating systems and energy management in complex distributed systems.

Researching voltage stability effects in power nets together with operating system level scheduling is largely an open issue. In the following, important related work is presented, touching aspects relevant to the problem statement of this thesis.

1.3.1. Power Management in Operating Systems

The aforementioned models abstract the underlying operating system and concentrate on the functionality of the application components. There exists a long history of related work in the field of power management in operating systems. As a modern and widely used operating system, the Linux kernel functionalities are briefly sketched.

For assigning resource budgets to trees of processes Linux, e.g., annotates its process tree with control information. Processes can be assigned bandwidth values or other fixed resources, which must be respected by the scheduler.
For the Linux operating system, different real-time application patterns are supported by different extensions such as those developed by Kim et al. (2002, 2005). These extensions target adjustable QoS settings.

For very low power wireless sensor networks, an assessment of efficiency of scheduling mechanisms is given by Mazumder et al. (2012). The assessment is special in regards to applications with large sleeping times in the order of seconds, and does not account for real time properties. The implemented algorithms are inspired by the Linux scheduling subsystem. More details on these operating system concepts are given in Chapter 2.

Quality of Service (QoS)

Related to power management is the topic of QoS. Waking up a hardware component from power saving mode, typically induces latencies. The same can apply to switching component frequencies, which may interrupt active tasks and deteriorate timer accuracy. During the switching of power states, requests can not be processed and thus deteriorate the QoS of the operating system platform to the applications.

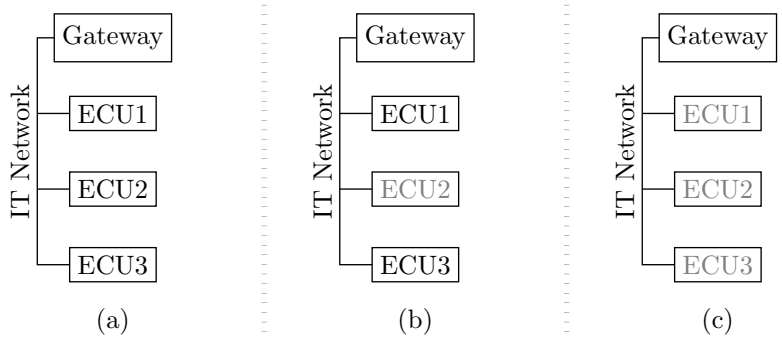


Figure 1.5.: AUTOSAR Mechanisms for Energy Efficiency in Electronic Control Units

Linux allows to adjust QoS requirements in different hard- and soft real-time categories. Hard real-time refers to latencies induced by hardware wake-up, while soft real-time measures resource usage in bandwidth, i.e., resource amount per time unit. These bandwidth measures can refer to CPU time per second, or communication throughput. Using Linux, one may additionally assign resource budgets using resource kernels, or starting from version 2.6, by using the integrated control groups data structures.

1.3.2. AUTomotive Open System ARchitecture (AUTOSAR)

The AUTOSAR includes mechanisms for turning on and off single Electronic Control Units (ECUs), virtual function clusters, and bus systems (AUTOSAR, 2013). Figure 1.5 shows different modes of operation enabled by AUTOSAR. In 1.5.a, all embedded systems are active and running to support different automotive functions. In 1.5.b, a selected system was at least partially shut down by a notification mechanism. If the operating system and computing units are still running, this is called degradation. When in pretended networking mode, all system components are shut down and the networking capabilities are loaded off to a dedicated piece of hardware, sustaining the operation of the communication network (Schmutzler et al., 2010). In this mode, periodic signals can still be sent from the system and specific input patterns may trigger a restart of the system.

Figure 1.5.c shows the feature of partial networking. In this mode, the complete bus system can be shut down. Thus, the instantaneous power consumption can be reduced to a minimum (Fuchs et al., 2010).
The feature of partial networking is standardized in a decoupling mechanism similar to publish/subscribe. The event service serves as the master component, with which publishers explicitly request the presence of a virtual function cluster. The requests are mediated in a way, that the subscribers delivering the functionality of the cluster receive explicit notifications to turn themselves on and off.

This decoupling is very helpful in coping with the complexity of the functions and to determine the necessity of components being online. During system runtime, the standard neglects the power consumption dynamics and supply voltage stability of the vehicle as an electric system.
These effects can be taken into account not only online, but also offline, by carefully

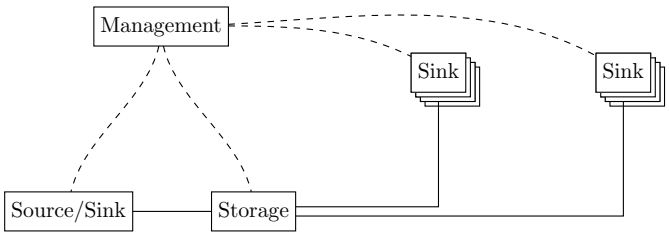


Figure 1.6.: Schematic Energy and Power Management

modeling the vehicle dynamics.

1.3.3. Energy Flow Management

Despite the view on electronic functions from a software perspective, there is a lot of related work regarding the electrical or energy flow in an electric system.
A vehicle can be seen as a network of sources, storage units, and sinks for electrical energy (Gehring et al., 2009). Figure 1.6 depicts such a network.
The network consists of at least one source of electrical energy, one storage component and different classes of sinks.

For managing this system of energy conversion components, cybernetic power distribution was proposed by Kohler et al. (2010, 2011) and subsumed in the PhD thesis (Kohler, 2013). The approach incorporates theory of cybernetic control, recombining logical and technical levels of the architecture.
There may be technical hierarchies due to the employed energy conversion processes. A group of energy sinks may be powered by a dedicated power supply line. This power supply line may be converting in between different voltage levels and may in its own depend on an extra source, like an alternator. Using the notion of technical hierarchy, arbitrary hybridization and energy distribution concepts can be described.
The logical hierarchy envisions a dedicated distribution master component in the architecture, which performs strategic decisions based upon predictions.

Electric Energy Sinks

In combustion vehicles, energy sinks are formed by assistance systems like chassis control, electric power steering or stability control, but also by all comfort and entertainment functions present nowadays.

In hybrid and electric vehicles, the drive-train poses an important component, which mostly serves as a sink, but also as a source of electrical energy upon recuperation.

The energy consumption dynamics of many such sinks may not be altered. This is due to safety requirements. Still, non-critical comfort devices may be altered, as well as the general process of controlling peripherals.

Electric Energy Sources

Depending on the type of vehicle, be it combustion, hybrid, or electric, different battery technologies are used (Ceraolo, 2000).

Battery technology has seen some progress in recent years. Combustion cars still use legacy lead-acid batteries together with an alternator for converting mechanical into electrical energy. While the lead-acid batteries feature highly dynamic capabilities due to a low inner resistance, the alternator is quite limited, because it has to be synchronized to the combustion engine.

Newer battery technology features larger capacities but worse dynamic capabilities. This can be compensated by including a mixture of battery technology or also by using capacitors to cope with transient peak power demand (Polenov et al., 2007).

Managing energy flows between vehicle components helps optimizing energy efficiency during operation.
For vehicle safety, transient processes have to be considered which leads to the topic of voltage stability.

1.3.4. Voltage Stability

There are established practices to design the topology of an automotive wiring harness. One typically has a battery and an alternator close to the engine. These components power a set of consumers over relatively long distances. The consumers can electrically be considered as loads imposed on the system.

In order to research into voltage stability, it is important to understand the dynamics of the involved components. Gehring and Herzog (2009) have researched into simulation models for stability in automotive 12 V power nets. Kohler et al. (2011) have built on these results and have conducted additional experiments in a test bench.

For safe operation, different types of electrical energy sinks may be supplied on different voltage levels. Operating on higher voltage levels enable loads to draw higher amount of power with less resistance. The power net may be supported by using stabilization units operating and converting in between different voltage levels.

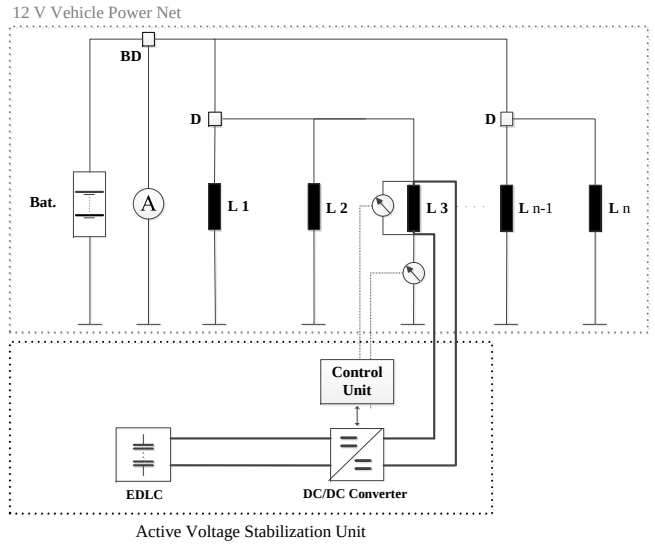
Stabilization Units

Transient loads in the electric system of the vehicle can lead to critical voltage drops. Figure 1.7a depicts a schematic of an automotive power net featuring an optional stabilization unit. Without the stabilization unit, significant voltage drops can occur in the wiring harness, due to the electrical properties of the wires. This is illustrated in Figure 1.7b. Due to the geometric nature and electric properties of the wires, the voltage drops in the system vary in significance.

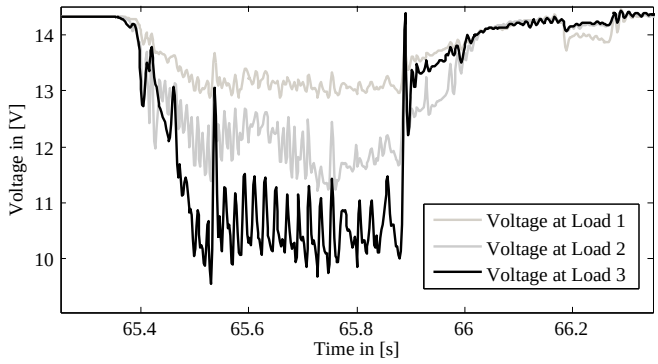
In order to prevent the voltage from dropping, stabilization units may be deployed along major loads. Figure 1.7c depicts the results of such a stabilization. The voltage levels for all three loads is significantly increased by supplying Load 3 out of a short term storage and energy conversion unit.

1.3.5. Precision Time Synchronization

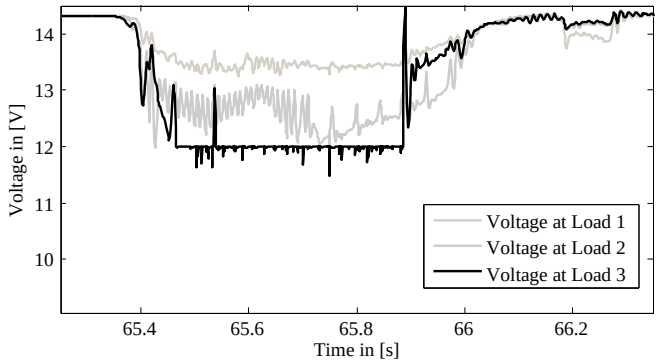
In the past decade, an IEEE standard for synchronizing clocks in distributed systems was crafted (IEEE1588). The standard was first implemented for the Linux operating system by Correll et al. (2005). They show that the level of synchronization reached by



(a)



(b)



(c)

Figure 1.7.: Automotive power net and voltage stabilization example. Figures taken from Ruf et al. (2013b)

this software-only implementation is quite good, provided the system is idle and may prioritize the protocol above all other tasks.

Since achieving good synchronization does not work well in software-only scenarios with high load, a lot of research was put into hardware assisted implementation of the IEEE1588 (Ohly et al., 2008). For supporting PTP hardware clocks, a new Linux Subsystem was designed by Cochran and Marinescu (2010). They are using the Subsystem together with hardware support, to later-on synchronize the Linux system time (Cochran et al., 2011). Kovácsházy and Ferencz (2012) compare the use of different hardware components to assist the synchronization process. Their results also indicate that the system load significantly deteriorates the quality and speed of the synchronization in software-only implementations.

Zhu (2013) compares Phase Locked Loop (PLL) and Frequency Locked Loop (FLL) implementations for the case of Base Stations as part of a telecommunication network. The experiment was conducted on a IEEE1588 capable processor. Improvements on software-only methods which rely on operating system mechanisms are largely an open issue, especially with presence of critical tasks and high system load.

1.3.6. Resource–Constraint Project–Scheduling Problem (RCPSP)

Besides scheduling for energy efficiency, as discussed by most previous work, one may want to ask for schedules which satisfy voltage stability criteria. The problem characteristic may be found similar to resource constraint project scheduling. The field of project scheduling stems from operations research (Colak et al., 2013). It is used to optimize the project in respect to different objective functions. These may be the total completion time, the number of resources used etc.

Multi-Mode

An extension to the simple RCPSP is to introduce multi-mode execution of jobs. This means an activity or job may be run in different modes which require different amounts of resources.

The traditional partitioning problem of software to subsystems may be tackled by introducing modes not only for operation modes of the underlying hardware, but also for the selection of a partition variant.

Voltage stable scheduling would mean to introduce additional constraints to the classic scheduling problems which may be described using the generalized notion of resources in project scheduling.

Resources

Resources in project scheduling may either be

- renewable, meaning at each time step, a set of resources becomes available. This kind of resources is typically labor, machines, and can be
- non-renewable, e.g. total budget for a project.

Scheduling in regards to voltage stability could be tackled using this framework by assigning a maximum amount of current draw for each time unit. The model lacks features of hard real-time scheduling problems such as end-to-end deadlines and cyclical execution within a set of jobs. These additional constraints are essential in regards to providing functional correctness of an embedded system.

1.4. Contribution

An execution model of tasks to be run in the operating system was crafted, which incorporates the active scheduling of hardware power states. This extended notion of scheduling is culminated in the definition of generalized Power Management Plans (PMPs). The formalism allows the flexible assignment of plans to arbitrary functional system states, thus transducing from system inputs to sequences of schedules.

The solution is described in a mathematical manner and implemented in prototypical embedded systems. To support the engineers in the early design-phase, assistance in model checking and exploration is given based on solving generalized constraint satisfaction problems. The problem formulation also regards constraints on voltage stabilization. Since voltage stability is a local phenomenon which in turn results from a global system behavior, time synchronization is important to enable scheduling for stability.

As a proof of concept, the methodology of PMPs is applied to a software-only implementation of the IEEE1588 Precision Time Protocol (PTP). Results show that PMPs enable the combination of critical system tasks with a lower prioritized synchronization process. The platform guarantees high QoS levels and additionally enables significant energy savings.

1.5. Structure

The structure of the work at hand is divided into two parts. The first part covers the concept and modeling of operating systems and distributed functions in Chapters 2–4.

Chapter 2 describes important operating system concepts which serve as the foundation of the concept and implementation presented in this document.

Chapter 3 mathematically defines the concept of an operating system mechanism which proactively transduces system inputs to sequences of Power Management Plans (PMPs).

Chapter 4 details an approach to construct and check power management model instances for correctness based on formal methods.

The second part copes with the implementation and the evaluation of the concepts in a prototypical testbench.

Chapter 5 describes the protoypical implementation of the concept in Linux version 3.0.

Chapter 6 gives experiment setups and evaluation results.

Chapter 7 concludes the thesis.

2. Operating System Concepts

Key operating system concepts include hardware abstraction, resource management, in- and output, security and scheduling. For the operating system to support power management, all these aspects are affected.

2.1. Operating System Structure

Operating systems are designed to intermediate between hardware, software and the user. To achieve this task, different concepts were developed in the past and are under investigation nowadays. To structure and compare different approaches, Figure 2.1 retypes the aspects relevant to this thesis. Similar and more general diagrams can be found, e.g., in Silberschatz et al. (2005).

Chapter 2 is organized around Figure 2.1, starting with generic operating system components and later-on details specific real-time concepts.

2.1.1. Hardware Access

One key task of the operating system is to allow for heterogeneous applications to run on and share computing resources. On the lowest layer of the operating system, the access of the underlying hardware is managed.

Modern processing hardware has different levels of privilege, of which only the highest levels allow direct hardware access. The actual hardware access is always mediated by the operating system. In the so-called Hardware Abstraction Layer (HAL), device drivers provide abstract descriptions of and interfaces towards the underlying hardware.

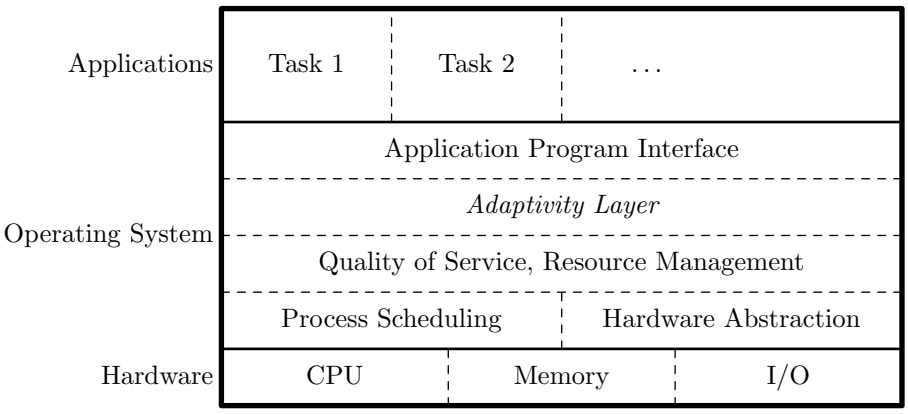


Figure 2.1.: Operating System Modularization

Computing resources such as memory and computation time are typically shared in a time and object division multiplexing manner.

2.1.2. Memory Access

For processes and memory access, different paradigms can be employed. In the following, the techniques of addressing memory in real and protected mode are presented

Real Mode Addressing

When programming hard real-time applications on low end hardware, resources often need to be accessed immediately and directly. For memory, this is called real mode addressing. In real mode, every process can access the whole physical memory directly. Since the access is very pure, switching processes has few overhead.

While the approach allows to fulfill very strict requirements, all knowledge has to be encoded into the application. This makes the solutions highly customized to the platform at hand. With increasing computing power and versatility of the employed hardware, the demand for flexibility of deployment and abstraction of hardware has also affected the real-time domain.

Protected Mode Addressing

Multipurpose operating systems running on stronger hardware introduce virtualization mechanisms. Application processes thus are typically isolated using hardware support for memory virtualization. Each process has its own virtual address space so another process' working set can not be accessed directly. Real-time tasks can be implemented using protected mode processes. Since processes may compete for access to resources, the management and locking of threads has to be kept in mind when running real-time tasks.

2.1.3. Resource Management

Not all resources can be shared. When a resource is exclusive, synchronization and locking mechanisms have to be used. Since these mechanisms in combination with unknown applications can lead to deadlocks, hard real-time systems rely on offline analysis of the involved system parts.

Locking

Depending on the type of computing hardware and resource to lock, different locking mechanisms have proven useful.

For short interval locking in uniprocessor systems, the code can temporarily disable interrupts. In this way, the code is uninterruptible and thus guarantueed to have exclusive resource access. Due to interrupt processing latencies, in- and output operations involving peripheral hardware can be disturbed.

This can be tackled by using spinlocking in multi core or multi processor environments. Spinlocks perform active polling and testing of resource availability. Like disabling

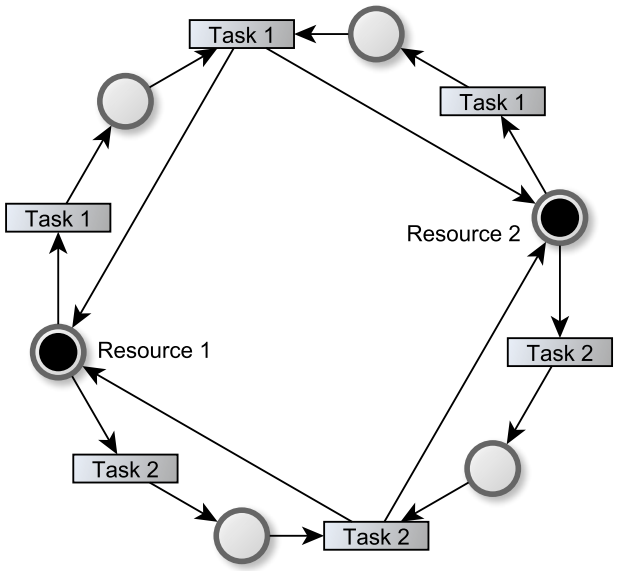


Figure 2.2.: Resource Processing in Tasks, Prone to Deadlocking

interrupts, it should also be used only in short time frames, because polling introduces unnecessary CPU load.

If a lock has to be held for a long time frame, complex mechanisms like Mutexes or Semaphores can help improve overall system performance. If a Mutex is held by a task and requested by another one, the requesting task is put into waiting state, marked as not runnable. Thus, it does not use CPU cycles to wait for a resource. When the Mutex is released by the first task, the second task gets into the runnable state again. The latency for dispatching the tasks and assigning the resource is higher than using spin locks.

The concept of Semaphores extends the Mutex paradigm by introducing a counter. The counter specifies the number of tasks to wake up. Typically, a producer pushes items into a queue, raises the semaphore to the queue depth, and thus releases up to as many tasks as there are work items.

Resource Allocation Analysis

The interplay of resource allocation and their timing can be formally analyzed. A resource allocation graph is a directed graph over resources and tasks, associating locked resources with tasks and tasks with requested resources. If the allocation graph contains a cycle, deadlocks are possible and would need to be worked around by using additional locking mechanisms. Figure 2.2 depicts an example of a petri net of two tasks competing for two resources which is prone to deadlocking.

The resource allocation and processing behavior can be analyzed using (timed) petri nets, or (timed) automata. These formal models provide timing information as well as the static information present in the allocation graph. Both timed models can be checked and proven for correctness. Correctness in this case is reachability of end states

without deadlocks, or exclusion of race conditions.

Measuring and affecting dynamic properties of computing systems is subsumed by the term of Quality of Service (QoS).

2.1.4. Quality of Service (QoS)

QoS manages adaptive performance guarantees to applications. This is done by monitoring and observing performance figures during run time. Because applications are encapsulated by operating system processes, establishing QoS is tightly related to process scheduling and the interaction in between hardware components.

The performance guarantees comprise the measures defined according to the following topics.

Latency

QoS requirements on latencies for processing events are a common measure in hard real-time systems. This latency can, e.g., relate to handling an interrupt request of a device, establishing a Direct Memory Access (DMA) process, or starting a real-time task at a given time.

For guaranteeing tight latencies, it might be necessary to restrict the deepness of sleep states of hardware, due to wake-up delays. An additional measure for time division multiplex resources is making tasks and resource locking preemptible where possible. Minimizing non-preemptible code paths can be done by deferring processing of large workloads.

Jitter is another QoS measure important for tasks to control physical processes. Jitter specifies the deviation of scheduling times of periodic tasks.

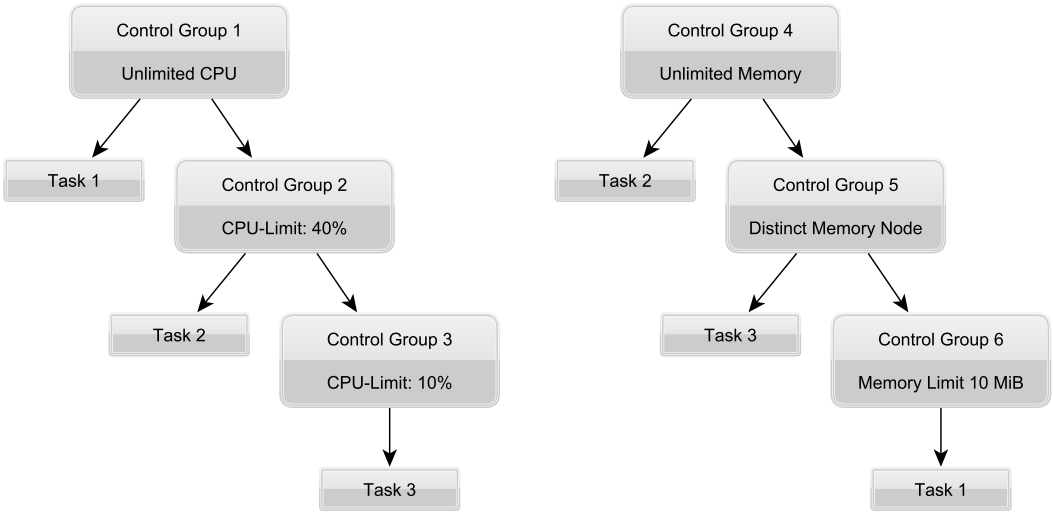
Resource Constraints

Additionally to latency, resources might be constrained by total budgets. This can be a total amount of a resource for the lifetime of an object, or a bandwidth, meaning constrained resource usage per time unit.

Figure 2.3 depicts two examples for hierarchical grouping and controlling of resources in the Linux operating system. The control groups can be annotated with constraints for resource allocation subsystems, like CPU accounting, or memory (Menage, 2006). Each control group can contain Linux tasks or yet more control groups.

In Subfigure 2.3a, three tasks are partitioned into control groups for CPU accounting. The root group features unrestricted resource usage for Task 1. Control group 2 restricts Task 2 and Control Group 3 to a combined maximum CPU limit of 40 %, of which Task 3 can amount at most for 10 %.

In Subfigure 2.3b, the same tasks are structured into different control groups annotated with memory allocation constraints. The root group again is unconstrained, containing Task 2 and Control Group 5. Group 5 is limited to memory allocation at a specific node of memory, i.e. a specific piece of hardware. This applies both to Task 3 as well as Gontrol Group 6, which additionally limits the total memory usage of Task 1 to 10 MiB.



(a) (b)

Figure 2.3.: Linux Control Group Hierarchies

Shaping

The implementation of QoS is typically done by a combination of mechanisms (Aurrecoechea et al., 1998). Resource usage can be affected by scheduling both computing tasks as well as in- and output. Using QoS aware scheduling leads to shaping of resource usage. This can be done by adjusting the schedule in a way which affects throughput. Scheduling of in- and output is necessary to synchronize applications in a distributed system.

Despite scheduling, the QoS can be affected by manipulating queuing mechanisms. This can be done by filtering of in- and output. Queues can also be influenced by adjusting maximum depths such as the window size in TCP.

Adapting resource usage and specifying QoS levels is done in the adaptivity layer of the operating system.

2.1.5. Adaptivity Layer

One important abstraction for safety and security is the separation of applications. This is established by using hardware privileges and virtual memory.

The ARM Cortex platform features several operating modes, which can be classified as `privileged` and `user` (ARM Ltd., 2010). Most notably, the operating system and device I/O happens in privileged mode and the processes running on the operating systems are running in user mode. The OS has to provide arbitration for requests from applications. This is typically the task of checking for (security) rights, or mediating QoS requests.

In this thesis, the adaptivity layer allows implicit as well as explicit statements. These types of statements are explained in the following.

Explicit Adaptivity

Explicit adaptivity can be an explicit state request for a resource. Including, but not limited to:

- Locking and sharing of resources, as well as scheduling of in- and output. Applications can exclusively access locked resources. This access can be used to explicitly control resources from within an application context.
- Control of resources may be used to explicitly switch power states of hardware. An example would be to have an application execute a user request to shut down a device.
  To do this kind of explicit configuration, the virtual device node representing a specific piece of hardware in the operating system, has to be opened and configured. The application doing this has to be equipped with appropriate rights and typically claims exclusive access to the resource. Thus, the management of the hardware shifts from the operating system to the application.
- Despite configuring peripherals, the applications may explicitly change operating system policies such as scheduling. This way, applications can configure mechanisms which exhibit implicit adaptivity. An example would be to tune QoS parameters which later-on get applied implicitly by the operating system.

Resorting to explicit adaptivity makes the configuration part of the application. If this configuration is spread over all applications, emergent system properties may be difficult to maintain and to verify. This is especially true if the partitioning of software components are being changed.
During the task of partitioning, the configuration of such applications would need to be adapted. This may be costly or even impossible.

Mixing different software units brings up emergent behavior as a combination of requests.
For the verification of correctness, either the exact behavior has to be specified, or, implicit approaches can be chosen.

Implicit Adaptivity

Implicit adaptivity is based on a rule set, translating in- and output of applications into actions, meaning for every state the adaptivity layer is in, an exact and explicit new state is known for each observed signal. That way, no arbitration time is needed and the superposition of requests can be analyzed offline.

Examples for implicit adaptivity may be:

- Enforcement of system correctness, like closing an application implicitly closes all its open files. This is related to the topics of safety and security.
- The routing of network packets in the operating system. An application typically just specifies an end-point and utilizes operating system services to handle the communication. Depending on the routing table, the operating system then implicitly handles address resolution and execution of the request.

- The execution of requests may be influenced by filtering rules and actions, such as used by the netfilter and iptables subsystems in Linux.

Resorting to implicit adaptivity implies equipping the operating system with the knowledge necessary to adapt to and execute the applications. This way, applications can be recombined, without changing or adjusting explicit requests.

2.1.6. Application Program Interface

The application program interface provides all means to make software interoperable among different hardware platforms. Due to the abstraction and translation of requests, the applications typically do not work directly on I/O hardware but use handles or tokens which represent a request.
Using the interface, applications can communicate with one another and the outside world. All these actions are governed by the implicit adaptivity features. Because of this governance, the interfaces provide a feedback mechanism, indicating the state of requests.

2.2. Real-Time Operating Systems

Real-time operating systems are typically lean and specialized to the task at hand, especially when running hard real-time tasks.

2.2.1. Hard Real-Time Tasks

Hard real-time tasks have strict scheduling requirements, typically specifying periodic execution relative to fixed time offsets. Figure 2.4 depicts an exemplary schedule with offset $o(j)$ and period $p(j)$.

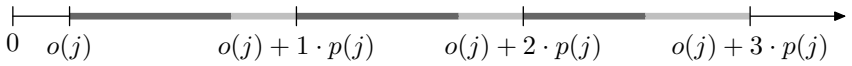


Figure 2.4.: Fixed Cycle Scheduling of Hard Real-Time Tasks (Barthels et al., 2011)

If the worst-case execution times and the time allocation for different tasks overlap, tasks can be preempted by other tasks with higher priority. If more than one task with the same priority is runnable at the same time, e.g. the Earliest Deadline First (EDF) policy comes into place.

Hard real-time tasks typically are used to control physical processes. Since these processes have certain time evolution, the processing has to take place at exact time points.
These time points may be specified relatively to one another. An example of this is the specification of end-to-end deadlines. Additionally to specifying the exact time for running a task, one can specify bandwidths of time usable by a system task in soft real-time.

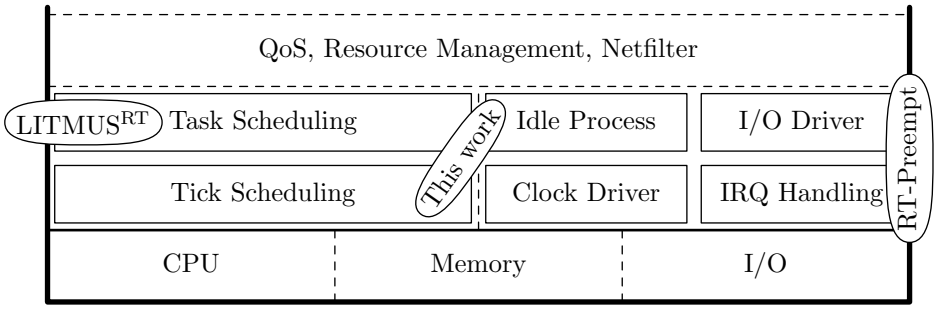


Figure 2.5.: Differentiation in LITMUS, RT-Preempt, and the Contribution of the Thesis at Hand

2.2.2. Soft Real-Time Tasks

Soft real-time tasks typically do not have to be executed by time of their deadline. They only need to have a guaranteed total resource time per time unit, i.e., bandwidth. These tasks do not need to rely on latencies or accurate sleep calls. Examples for soft real-time tasks include computationally intensive processing of bulk data such as in image processing.

2.2.3. Implementation Concepts

For establishing hard real-time timing properties, the I/O handling, locking, and interprocess communication is handled by small kernels which can e.g. be loaded as resource kernels into Linux. This approach is used by Xenomai (2013) and Bucher et al. (2013). Early work on resource kernels can be found, e.g. in Oikawa and Rajkumar (1998).

In recent years, the Linux kernel itself has seen strong development. More recent contributions, enrich the Linux kernel itself with features targeted at real-time processing. The kernel was, e.g., extended with an EDF–scheduling class by Faggioli et al. (2009) and by Calandrino et al. (2006) as part of the LITMUS^{RT} environment. The approaches work on top of the task scheduling mechanism and provide a new scheduling discipline. Both approaches do not take energy awareness into account. Another train of work is to improve scheduling latencies using the RT-Preempt patch developed by Molnar (2013). The patch introduced nested and threaded interrupt handling, as well as preemptability of the kernel itself. These features were by now integrated into the kernel. The patch nowadays is a collection of changesets which mostly apply to device driver subsystems, making these preemptible were possible.

Figure 2.5 depicts the different points of application of recent related work in the Linux kernel. RT-Preempt is located on the right side, affecting interrupt handling and I/O drivers for minimizing system latencies. LITMUS^{RT} is a framework for researching into task scheduling policies. For a comparison of LITMUS^{RT} and RT-Preempt, please refer to Cerqueira and Brandenburg (2013).

The thesis at hand also touches task scheduling, but focuses more on energy awareness by adjusting system time slices and the system idle process.

Very small embedded systems typically do not run Linux as an operating system, since it is quite complex and may be expensive to run.

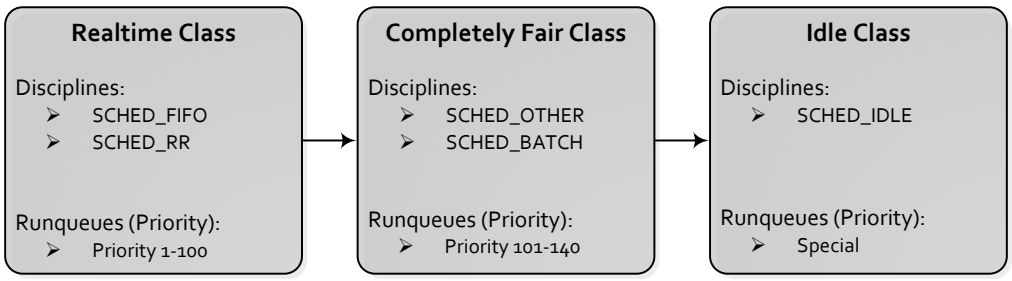


Figure 2.6.: Scheduling Classes as Being Implemented in Linux 3.0

Especially for embedded systems with limited resources, lean operating systems were developed, such as TinyOS (Levis et al., 2005). These lean operating systems handle only the smallest part of the system with leveraged privileges (kernel mode). This approach is most often referred to as Microkernel.

Despite scheduling and I/O, all other aspects of operating systems is handled in user mode. This processing mode is tightly coupled to threading and processing abstractions. These operating systems improve on timing guarantees in event handling on very small devices. Tasks are being higher prioritized as the operating system itself. Due to the constant increase in computing performance of the hardware, this thesis concentrates on Linux as a platform, which has seen increasing adoption in the field.

2.3. Process Scheduling

The system scheduling in modern operating systems has a profound number of aspects. Figure 2.6 depicts the scheduling classes used in the Linux kernel, as they supersede each other in priority.

Operating systems provide means of synchronization. One can synchronize to external events (interrupts), or tasks can synchronize themselves using local mechanisms and using the system scheduler.

Processes can be in waiting state. When in this state, they are not being scheduled to use the CPU. A process can be woken up by any event at any point in time. The overall system behavior then strongly depends on the current scheduling paradigm.

2.3.1. Preemption

There are different forms of preemption. Preemption may be disabled, which in its strict sense also includes disabling interrupts. Preemption may be induced by timer interrupts, or also in general by locking mechanisms, which can change a task's state.

In Linux, e.g., tasks are in general not preemptible while executing code in kernel space. The kernel can be explicitly configured to allow this. Still, preemption cannot occur, when the task holds specific locks or has interrupts disabled. There are patches

to Linux, like RT-Preempt (Molnar, 2013), which ensures preemptability of tasks in almost all cases.

2.3.2. Queuing

In Linux, each priority of each scheduling class has its own queue of tasks. Runnable tasks are removed from the queues in the order of their priority and are reinserted upon yielding the processor depending on the queuing discipline.

First-In First-Out (FIFO)

The FIFO discipline is the simplest discipline, which would typically be applied to hard real-time tasks. FIFO tasks are executed in the order in which they arrive. They are not being preempted by tasks of identical priority. Instead, they can run until yielding the CPU to the next task in the queue.

Round-Robin (RR)

Using the RR discipline, all tasks in each queue are multiplexed in a time division manner. In this way, even when all tasks are runnable all the time and there are fewer CPUs than tasks, all tasks with the same priority still get a share of CPU time.

Completely Fair Queuing (CFQ)

The average number of runnable tasks over time per CPU is defined as the CPU load. This load equals the depth of the queue of runnable tasks.

CFQ extends the notion of load to a task wise tracking. CFQ uses the exact measure of nano seconds each task has run since its creation. It aims to exactly even out the amount of runtime per task.

As in RR, this is established by using a time division manner.

2.3.3. Time Slices

The time allocated to a runnable task is sliced, meaning tasks are interrupted regularly. Upon this interruption, the computing times of the processes get accounted and the tasks are scheduled. This scheduling is done according to the aforementioned policies. In Linux, the interrupt is called the system tick, marking the end of the current time slice and the start of the next. Fedorova et al. (2007) have investigated into the effect of choosing time slices in order to do load balancing among threads with equal priority on heterogeneous multi-core systems. In contrast, this thesis focuses on power management and scheduling of tasks with different priorities on lightly loaded distributed systems.

When a task of higher priority is runnable, all tasks of lower priority have to wait. Upon equal tasks with equal priority, sophisticated schemes were developed to allow for a fair distribution of computing time. Finally, in the idle class, system dependent idle code is executed, establishing partial system degradation and managing system wake up.

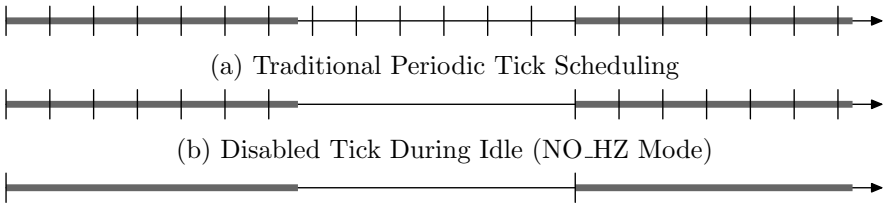


Figure 2.7.: Comparison of current Linux tick scheduling. Gray intervals indicate busy times on the computing hardware. In NO_HZ mode, the regular ticks are disabled by the idle task.

Unfortunately, due to this degradation and wake up, time is lost. This can be compensated by a novel approach to tick scheduling which is being elaborated in Chapters 3 and 5.

2.4. Timing Abstractions

The system tick denotes the event upon indicating the end and start time of a system time slice. Traditional system tick or time slicing was provided by periodic clock sources. They were programmed to provide an interrupt signal at strictly cyclical intervals. Figure 2.7a depicts such a traditional tick period. In this case, ticks occur both when tasks are running (gray intervals), and when they are not running.

Over the traditional approach, two improvements were developed and implemented for the Linux kernel.
The first is called NO_HZ and is illustrated in Figure 2.7b. It allows the system to reprogram the clock-device if possible to skip the periodic wake-up caused by ticks which are not needed for task slicing.
The second improvement is still in draft status and additionally allows to stop the tick during special tasks which are known to not be preemptible by other tasks (cf. Figure 2.7c).

2.4.1. Clock Synchronization

As a means for synchronizing clocks in embedded systems, the IEEE 1588 Precision Time Protocol (PTP) is used in this thesis. It defines a way to synchronize clocks in a distributed system to a selected master clock. In Linux, the synchronization works by providing a kernel time abstraction layer to the actual clock component. The kernel time can be sped up, slowed down or be reset to some arbitrary value. Because of this, the scheduling subsystem does not use this time base but directly operates on the system's hardware clock source.

For experimentation, the open source Precision Time Protocol daemon (PTPd) is used. The underlying concept of the protocol and daemon were published by Correll et al. (2005). The essential timing definitions are re-elaborated here. The algorithm to select a master clock is not described, it is assumed that all roles in the system are clearly assigned, having a single distinguished master component.

A clock hardware component is assumed to be counting intervals of elapsed time as large as the clock precision value $\hat{p}_c$ of clock c. Thus, a clock exhibits a natural number of intervals, i.e. clock cycles, measured in $\mathrm{TSC}(t)$ at any given time t. For simplicity, and for describing the PTP, real valued clocks are assumed. The real time is denoted t, a clock value $c(t)$ at a given time t incorporates an error term:

$$c(t) := \mathrm{TSC}(t)\hat{p}_c = t + \mathrm{err}(t),$$

where

$$\mathrm{err}(t) := \mathrm{skew}(t)t + \mathrm{offset}(t) + \theta(t)$$

with a nonlinear error term $\theta(t)$ respecting

$$\theta(t) = 0, \quad \theta'(t) = 0.$$

The PTPd uses a proportional integral controller to discipline the clock so as to minimize the error. The frequency of $c(t)$ is adjusted using an estimate of the error $\widehat{\mathrm{err}}(t)$.

$$\mathrm{adjust}(t) := a_P \cdot \widehat{\mathrm{err}}(t) + a_I \cdot \int_0^t \widehat{\mathrm{err}}(t)\mathrm{dt}.$$

with dampening factors a_P and a_I.

Since in a PTP system, there is no perfect clock, the error of clocks is estimated relatively to each other. Each clock, including the master clock, is assumed to have a linearized error at each time t. Thus, each clock is approximated as a linear distortion of the master clock $c_m(t)$.

$$c(t) := \mathrm{skew}(t) \cdot c_m(t) + \mathrm{offset}(t)$$

Both $\mathrm{skew}(t)$ and $\mathrm{offset}(t)$ may vary over time due to physical processes such as thermo- and electrodynamics within the circuitry. Thus, the estimation of offset and skew has to be repeated and adjusted appropriately over time.

This synchronization is two-fold, in the first step, the offset is minimized and in the second step, the rates of the clocks get aligned. Figure 2.8 depicts the two step interleaving processes for determining skew and offset.

Subfigure 2.8a shows the interaction from master to slave. On predefined intervals, PTP SYNC packets are sent from the master to the slave. If the hardware supports accurate time stamping upon sending, the packet may contain the time point of sending t_1 in its own. With lack of hardware support, t_1 is delivered as accurately as possible within a FOLLOW UP message. Repetitively sending these packets may help to keep the skew of the clocks in line, while the clients still need a measure of the one way communication delay in order to estimate the clock offset.

Subfigure 2.8b depicts the sporadic process of measuring the end to end communication delay by the slaves. Denote t_2 the time at which SYNC is received at the slave (multicast). The communication delay is assumed to be symmetric. The slave sends a

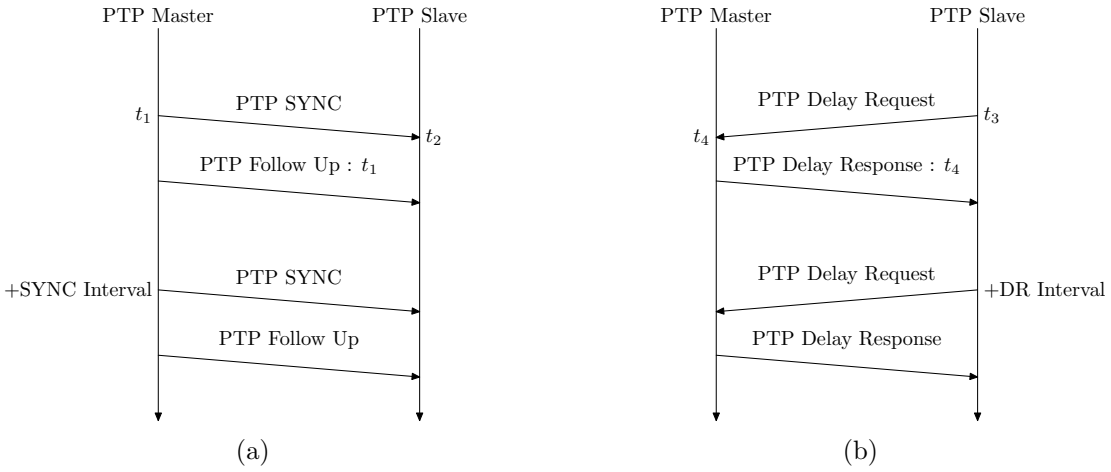


Figure 2.8.: PTP Protocol Flow as Depicted in Weibel (2009)

DELAY REQUEST and notes its own send time t_3. The master answers with a most accurate receive time t_4 which is transmitted as part of a DELAY RESPONSE message.

The relative communication time in between master and slaves, measured by their respective clocks is defined as:

$$\begin{aligned}\text{delay}_{\text{m2s}} :=& (c(t_2) - c_m(t_1)) = \text{skew}(t_2) \cdot c_m(t_2) + \text{offset}(t_2) - c_m(t_1),\\ \text{delay}_{\text{s2m}} :=& (c_m(t_4) - c(t_3)) = c_m(t_4) - \text{skew}(t_3) \cdot c_m(t_3) - \text{offset}(t_3).\end{aligned}$$

The one way communication delay is then estimated as:

$$\text{delay}_{\text{comm}} := \frac{\text{delay}_{\text{m2s}} + \text{delay}_{\text{s2m}}}{2}$$

After several sync messages the skew is assumed to be eliminated, skew $\equiv 1$:

$$\approx \frac{c_m(t_2) - c_m(t_1) + \text{offset}(t_2) + c_m(t_4) - c_m(t_3) - \text{offset}(t_3)}{2}$$

Assuming the offset to be constant within the interval between t_1 and t_4:

$$\approx \frac{c_m(t_2) - c_m(t_1) + c_m(t_4) - c_m(t_3)}{2} = \frac{\text{RTT}}{2}.$$

This approximates half the round-trip time (RTT) in the slave as it would be measured using the master clock c_m.

Using the communication delay, the offset from the master is estimated by

$$\text{delay}_{\text{m2s}} - \text{delay}_{\text{comm}} = \frac{\text{delay}_{\text{m2s}} - \text{delay}_{\text{s2m}}}{2}$$

using skew $\equiv 1$ and constant offset:

$$\begin{aligned} &= \frac{\big(c_m(t_2) - c_m(t_1) + \text{offset}\big) - \big(c_m(t_4) - c_m(t_3) - \text{offset}\big)}{2} \\ &= \frac{\big(c_m(t_2) - c_m(t_1)\big) - \big(c_m(t_4) - c_m(t_3)\big)}{2} + \text{offset} \end{aligned}$$

Assuming perfectly symmetric communication times, the nominator of the fractional term cancels and yields:

$$\text{delay}_{\text{m2s}} - \text{delay}_{\text{comm}} \approx \text{offset}.$$

The software implementation uses filtering and control theory to estimate and adapt the offset and skew over time. The better the predictability of the involved components, the faster the convergence. This thesis investigates how a power management planning scheduler can help to support convergence of the PTP in a distributed system.

2.5. Summary

This chapter presents operating system principles fundamental to embedded and real-time systems. It is promising to combine classical operating system tick scheduling with real-time operating system features to form a power management mechanism operating on the tasks at hand. The next chapter mathematically defines such a mechanism over tasks.

3. Logical/Technical Modeling

The abstractions presented in this thesis are based upon previous work. This chapter elaborates a model for sequence based power management. As a contribution of the thesis, this model is implemented in the Linux operating system. The implementation is detailed in the second part of this thesis.

In this chapter, the model description is guided along the layered operating system architecture presented in Chapter 2.

3.1. Cyber-Physical Systems

The class of cyber-physical systems was crafted recently. Figure 3.1 depicts the logical interaction scheme as presented in Barthels et al. (2011).

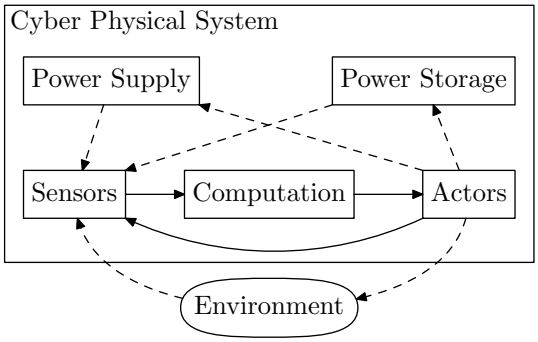


Figure 3.1.: Logical interaction scheme of cyber physical systems. Solid lines represent data communication links, while dashed lines represent physical processes.

The interaction scheme is assumed to involve Sensors, Computation, Actuators, and the respective Power Supply and Power Storage. These components all communicate in the directions shown with solid arcs in Figure 3.1. Additionally, physical processes are affected by the actuators and measured by the sensors.

3.1.1. Technical Abstraction

Next to the logical interaction, Barthels et al. (2011) give the following formal definition for the technical architecture of cyber physical systems:

Definition 1 (*Cyber Physical System*)

A cyber physical system $\mathcal{S}$ consists of sub-systems $\mathcal{E}$ which are interconnected by different network segments $\mathcal{N}$, a wiring harness $\mathcal{W}$, power supply and energy storage units $\mathcal{C}$, $\mathcal{S} = (\mathcal{E}, \mathcal{N}, \mathcal{W}, \mathcal{C})$.

The definition specifies the sets of technical architecture elements, on top of which the interaction scheme from Figure 3.1 is run. Specifically, communication network segments and a wiring harness are added.

The overall technical architecture is formed by interconnected subsystems, which again can be refined as:

Definition 2 (*Cyber Physical Subsystem*)

A subsystem $E \in \mathcal{E}$ is a quintuple $E = (\mathtt{A}, \mathtt{S}, \mathtt{C}, \mathtt{I}, \mathtt{P})$ of actuators $\mathtt{A}$, sensors $\mathtt{S}$, computational units $\mathtt{C}$, network interfaces $\mathtt{I}$, and power supplies $\mathtt{P}$. There must be at least one computational unit, one network interface and one power supply on a subsystem, and there can be arbitrary many actuators and sensors. Each subsystem E also comes equipped with a specification of the minimum supply voltage level $U_{\min}(E)$ it can operate at.

These subsystems form a distributed system which supports the logical interaction scheme as depicted in Figure 1.4.

3.1.2. Software Abstraction

The subsystems interact with each other and the environment using in- and output features. The interaction is run by the applications, and managed by the operating system. Figure 3.2 shows the components relevant to this thesis and highlights the application layer which is explained in the following subsection on software abstraction.

The power management concept is tightly interconnected with the application layer. The application layer is comprised of schedulable units, called Jobs, providing in- and outputs. Jobs correspond to the Linux task entities as depicted in Figure 3.2.

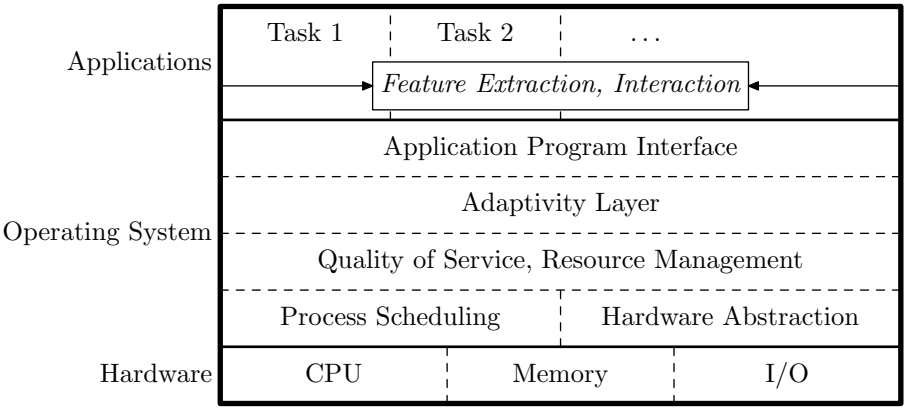


Figure 3.2.: System features are delivered by the application layer. Complex know-how and algorithms are involved and need abstractions.

The jobs provide the system functions and features, by utilizing possibly complex interactions across subsystems in the application layer.

Definition 3 (*Job*)

A job $j \in J$ is a template for an action within the cyber physical system $\mathcal{S}$. The following types of jobs are valid:

- Sampling sensor input:
  This job yields outputs interfacing processing jobs or actuator driving jobs.
- Processing:
  A processing job has at least one and possibly multiple inputs and outputs each. Processing typically means aggregation and state estimation as an input for the next processing or acting job.
- Acting:
  Acting jobs only have inputs and can incorporate two types:
  1. Acting as a self-parameterization of the system (Power Supply, Sensors, ...)
  2. Acting as driving an actuator to perform manipulation of the environment.

Each job is considered having requirements on cyclic scheduling, meaning they feature a cycle period p, a cycle offset o, and a deadline d

$$p, o, d : J \to \mathbb{R}_{>0}$$

Since each job must be runnable, it must include computations, network communication and control properties of sensors and actuators.

From the operating system or system integration standpoint, jobs can be treated as black boxes. In the automotive industry, e.g., jobs contain know-how of suppliers, and often only the interaction and features are specified.

Complex interaction patterns can be realized by communicating jobs, which form functional chains.

Definition 4 (*Functional Chain*)

A functional chain F is a weakly connected directed Graph $F = (J_F, E_F)$ linking jobs J_F with edges E_F.

One may additionally restrict these graphs to be acyclic, and to adhere to the interaction paradigms as in Subsection 1.1.3.

Since a functional chain itself needs an implementation, it has to be equipped with mapping information, making it possible to assign job instances to hardware elements.

Definition 5 (*Mapped Functional Chain*)

A *mapped functional chain* over a cyber physical system $\mathcal{S}$ is a functional chain F together with a function $\mathcal{M}_F$ mapping jobs to components of cyber physical subsystems $\mathcal{M}_F : J_F \to \mathtt{A} \cup \mathtt{S} \cup \mathtt{C}$. Sensor input jobs are mapped to specific sensors, processing jobs to computational units, and actuator driving jobs to specific actuators.

Being supplied with a mapping, power and performance values can be determined or at least bounded. Performance values relate to busy times of the underlying hardware until the completion of the jobs. Related to this timing, is the specification of deadlines.

Definition 6 (*End-to-End-Deadline Specification*)

End-to-End-Deadlines are specified by a relation $D \subset J \times J$ and by assigning a unique deadline time value $\mathcal{D} : D \mapsto \mathbf{t}_{\text{deadline}}$. Since each job j is already assigned its own

deadline value $d(j)$, the End-to-End-Deadline is defined as the time between the end of the earlier and the beginning of the later job.

Having specified deadlines, mappings can become infeasible to schedule. This way, scheduling and mapping of jobs is tightly related. How both valid configurations for mapping as well as scheduling can be determined is explained later.

Functional Hierarchy

To be able to turn on and off different functions, additional relationships, such as general feature interaction have to be defined.

Definition 7 (*Function*)
A function $\mathcal{F}$ consists of a set of mapped functional chains $\mathcal{F} = \{(F, \mathcal{M}_F)\}$. A functional chain may be contained in multiple functions. A function $\mathcal{F}$ is called active, if and only if all its chains are active, meaning all contained jobs are being running and scheduled according to their requirements.

Relating functions with one another in tree-structures yields hierarchic functions.

Definition 8 (*Hierarchic Functions*)
A set of hierarchic functions $\mathcal{H}$ is a forest of functions $\mathcal{H} = (\bigcup \mathcal{F}_i, G)$.

Using hierarchic functions, one can define meta-functions and use cases, like driving or parking. These functions then include sub-functions, which in turn may have more complex relationships with one another.

Arbitrary relationships can be introduced to form related functions.

Definition 9 (*Related Functions*)
A set of related functions $\mathcal{R} = (\bigcup \mathcal{F}_i, G + R)$ is a set of hierarchic functions $(\bigcup \mathcal{F}_i, G)$ together with additional directed edges R indicating relationships in between functions.

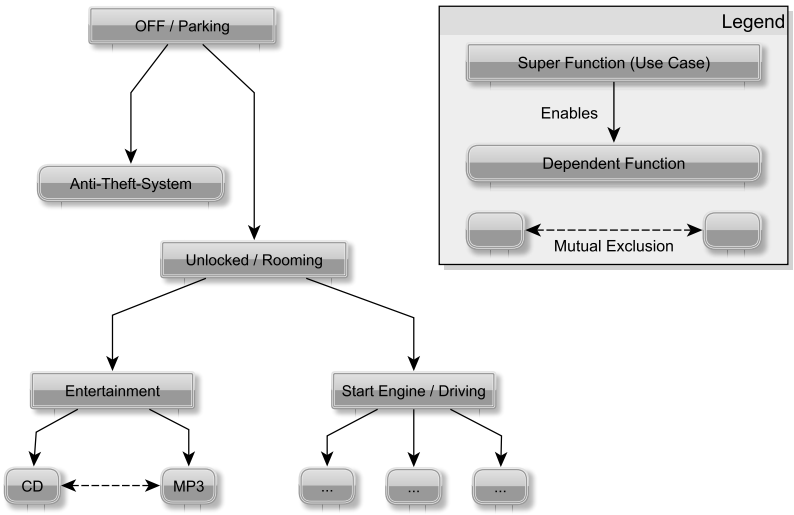


Figure 3.3.: Exemplary Functional Hierarchy Together with Mutual Exclusion

For an extensive discussion of function, service or feature relationships the reader is referred to Rittmann (2008). An example for related functions is depicted in Figure 3.3. It shows functions which are hierarchically dependent on one-another and functions which additionally exclude each other.

The functional hierarchy can be used for distinguishing sets of jobs necessary to each system situation. If the car is parking, e.g., the features exclusive to driving need not be supported and thus neither scheduled nor powered.

3.1.3. Power Management Planning

The power management planning is based upon Barthels et al. (2011) and was extended in the direction of a sequential logic approach as part of an implementation in a simulator (Gabriel, 2012). Figure 3.4 depicts the operating system layer relevant for executing PMPs.

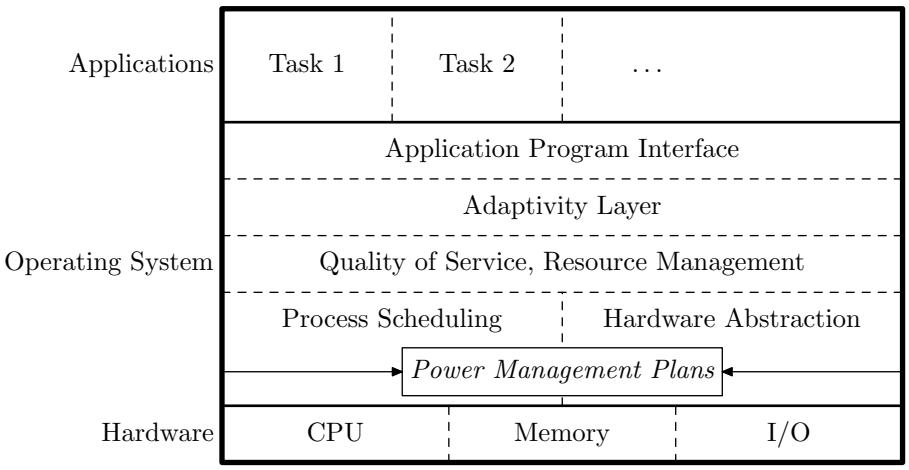


Figure 3.4.: Low Level Scheduling of Software Threads and Power (→ Hardware Abstraction) by Means of Power Management Plans

The definition of power management plans is built around sequential signal logic, which include the notion of signal timing explained in the following definition.

Definition 10 (*Signal Timing*)

Let $\mathbf{t}_{\text{signal}} := \mathbf{t}_{\text{set}} \times \mathbf{t}_{\text{clear}} := \mathbb{R}^2_{\geq 0}$ be the space of signal behavior timings. t_{set} delays the setting of an output signal after the setting of the respective input signal. t_{clear} delays the unsetting of an output signal after the unsetting of the respective input signal.

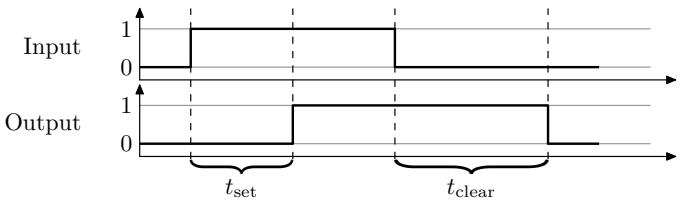


Figure 3.5.: Signal Timing, Set and Clear Properties

Figure 3.5 illustrates the signal level induced by the timing parameters $t_{\text{set}} \in \mathbf{t}_{\text{set}}$ and $t_{\text{clear}} \in \mathbf{t}_{\text{clear}}$.

Combining timed signals with jobs and hardware power states into a graph model yields the notion of PMPs given in the next definition. PMPs extend the traditional notion of scheduling in the operating system towards adaptive time slices and hardware power states in the operating system.

Definition 11 (*Power Management Plan*)

Denote a power management plan for subsystem E by $\gamma_E \in \Gamma_E$. γ_E is a graph linking jobs $\mathcal{M}_F^{-1}(E)$, power states A_E and control flow nodes together with their temporal sequence in signal timing behavior.

$$\begin{aligned} \gamma_E &= (C, H), \\ \text{type} : C &\to \mathcal{M}_F^{-1}(E) \cup A_E \cup \left\{ \bigcirc, \circledcirc, \ominus, \circledwedge, \circledvee, \circlearrowright \right\} \\ H &\to \{\text{Concurrent}, \text{Sequential}\} \times \mathbf{t}_{\text{signal}} \times \mathbb{1}_{\text{reset}} \times \mathbb{1}_{\text{periodic}}. \end{aligned}$$

Each edge $h \in H$ implements signal logic behavior, implementing signal levels l_h over time.

$$l_h : \mathbb{R}_{\geq 0} \to \{0, 1\}, \text{ for all } h \in H. \tag{3.1}$$

Control Flow Nodes

The control flow nodes have special syntax and semantics.

$\bigcirc$: **START** has in-degree of zero and out-degree greater zero. The outgoing edges get signaled exactly at the point in time of plan start.

$\circledcirc$: **STOP** has in-degree of one and out-degree zero. If the ingoing edge is signaled, is is possible to stop the plan, once requested from the system.

$\ominus$: **NOT** has in-degree one and out-degree greater zero. The outgoing edges are signaled exactly when the ingoing edge is not.

$\circledwedge$/$\circledvee$: **AND/OR** have in-degree and out-degree both greater zero. The semantics is an `and`, or `or` operator over all ingoing signal levels respectively.

$\circlearrowright$: **RESET** has in-degree two and out-degree one. Reacts on parameter $\mathbb{1}_{\text{reset}}$. Choosing parameter 1, sets and holds the output edge, even if the input is pulled down again. Choosing parameter 0 resets the output and it stays pulled down.

These control flow nodes allow to incorporate timed signal logic with scheduling of jobs J and transitions in between power states A_E.

Useful Plan Patterns

The formalism of PMPs enables a variety of scheduling options as depicted in Figure 3.6. One noteworthy example is the introduction of critical sections in order to gracefully

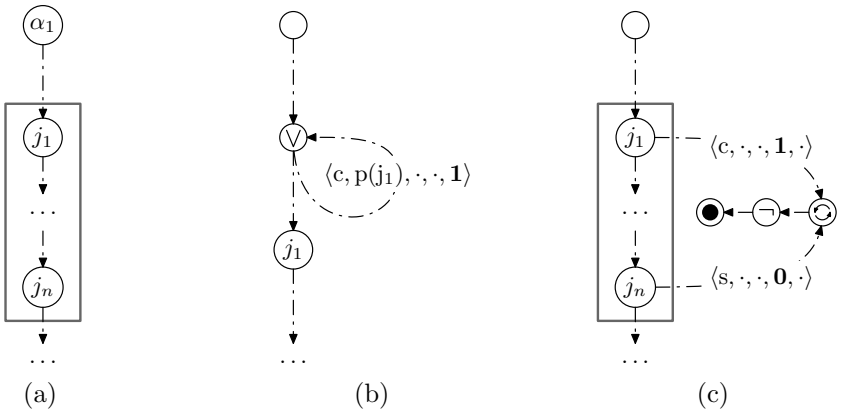


Figure 3.6.: Useful plan patterns for modeling common scheduling tasks. Figure a shows the ability to chain together job instances and choose the necessary power state for the series of jobs $j_1, \ldots, j_n$ beforehand. Figure b models the notion of a cyclically scheduled job j_1. Figure c builds a critical section $j_1, \ldots, j_n$, within which switching in between power management plans is prevented.

terminate and switch in between plans. Figure 3.6c shows the construction of such a section.

For different types of tasks and requirements, different types of patterns can be necessary.

Unrolling

Strictly cyclical schedules can be crafted using PMPs as depicted in Figure 3.7. Such PMPs can be uniquely unrolled onto a timeline including the dependencies of jobs and time of the PMP edges.

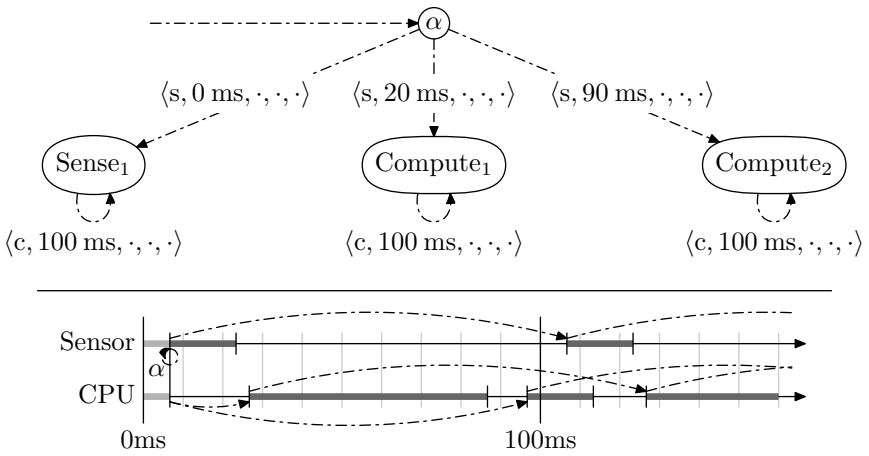


Figure 3.7.: Strictly Cyclical Jobs in Power Management Plan and Timeline View

For PMPs employing a mixture of time and event driven evolution, the graph structure can be plotted as a family of timelines. The evolution depends on the conditions in the

plan, which can relate to the behavior of the jobs.
The jobs are fragmented with idle times. For energy efficiency, prolonged idle times, allowing deeper idle states are desirable in semi utilized systems, as they are common in hard real-time systems.

3.1.4. Transducing Mechanism

Figure 3.8 shows the operating system architecture with highlight on the adaptivity layer. The adaptivity layer provides a transducing mechanism. This mechanism is defined in the following definition as the *power management module*. The module maps sequences of system inputs to sequences of power management plans. It was crafted in Barthels et al. (2011).

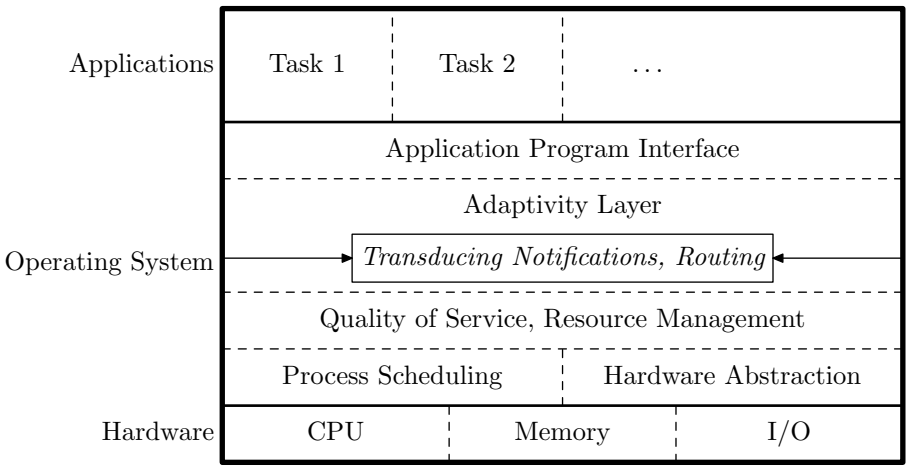


Figure 3.8.: Adaptivity is Modeled as Transducing Inputs to Plans

Definition 12 (*Power Management Module*)

Consider each power management module PM_E in each subsystem $E \in \mathcal{E}$ a transducing finite state (i.e. *Moore's*) machine (Moore, 1956):

$$\mathrm{PM}_E = (\Sigma_E, \Gamma_E, S_E, (s_0)_E, \delta_E, \omega_E),$$

where Σ_E is the set of receivable input characters, Γ_E is the set of power management plans, and, S_E is the set of functional states. Let $(s_0)_E$ be the initial state, δ_E be the transition function

$$\delta_E : S_E \times \Sigma_E \to S_E,$$

and ω_E be the output function

$$\omega_E : S_E \to \Gamma_E.$$

System Inputs

The set of receivable input characters Σ_E is chosen to be the union of all in- and outputs of jobs mapped to the subsystem $E \in \mathcal{E}$, $\mathcal{M}_F^{-1}(E)$. Thus, the system can react based on

the behavior observable in the operating system. This allows to support both explicit and implicit adaptivity.

Power Management Plan Set

Γ_E is the set of power management plans for the subsystem $E \in \mathcal{E}$. Each power management plan structurally relates jobs with power states and their timing conditions. How to construct valid plans, fulfilling functional requirements by the activated functions is discussed in Chapter 4.

Functional Status

A set of required functions is associated with each functional state $s \in S_E$. The functions are assumed to be hierarchically structured as in Definition 8. In that sense, the activity condition of a function is a generalized regular expression.

State Machine Decomposition

Transducers can be hierarchically decomposed using the notion of related functions explained in Definition 9.

The decomposition is a way to cope with model complexity. It is, e.g., present in the UML state machine superstructure model as defined by the Object Management Group (2011).

3.1.5. Response Flexibility

Activating the associated plan with a functional state is a reflex for providing system functionality. Typically, there can be many plans fulfilling a function requirement.

Complexity Reduction

One may reduce the complexity, i.e. the number of functional states tracked in the operating system, by limiting the variety of responses. Figure 3.9 depicts such a process of minimizing the size of a transducer by reducing it to its smallest equivalent. Step 3.9a depicts the original situation, reacting with three different plans to three intervals of vehicle speeds v. Finding a new plan γ_4 in Step 3.9b which both respects the requirements for state s_2 and s_3, may lead to the reduction into two states in Step 3.9c.

Energy Distribution

Altering the way how things are done, may influence energy distribution along with the scheduling of power consumption.

This is affected by implementing it as part of a cybernetic control approach, which is briefly described next.

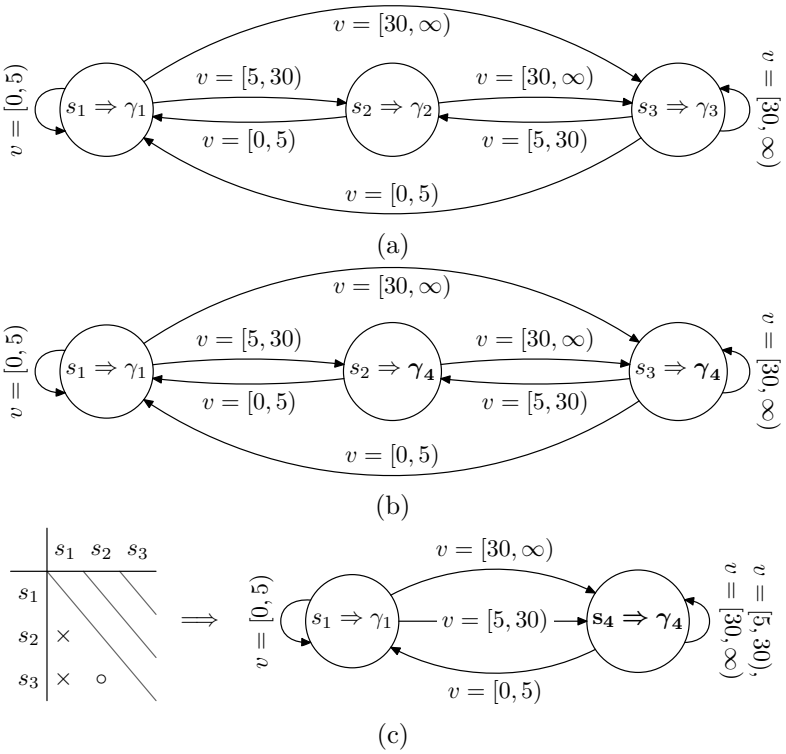


Figure 3.9.: Management complexity reduction through unifying plan generation. Illustration taken from Barthels et al. (2011)

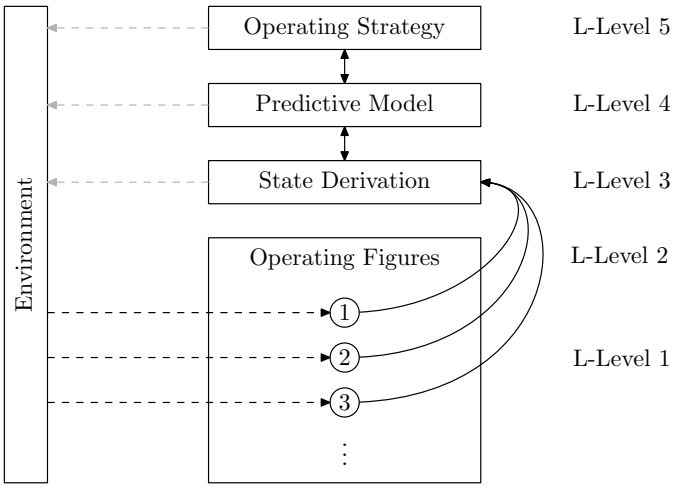


Figure 3.10.: Illustration of cybernetic control systems. Dashed lines are physical processes, solid lines are internal communication. Depicted are black arcs showing forward links, which are involved in deriving states, predictions, and strategies. Taken from Barthels et al. (2012a).

3.2. Cybernetic Control Approach

For energy distribution management, as well as voltage stability in automotive power nets, cybernetic approaches were investigated by Kohler (2013). Figure 3.10 shows the interactions of physical processes with internal communication and the top-down propagation of behavioral strategies.

3.2.1. Logical Levels

The internal interaction patterns are implemented within a logical cybernetic control hierarchy. On the lowest level, L-Level 1, sensor data is gathered. This data is processed to form Operating Figures in L-Level 2. Based on the Operating Figures, L-Level 3 adds derivation of semantic system status, which reflects the overall condition of different technical subsystems. The combination of subsystem states may be used for predictions and strategic decisions in L-Levels 4 and 5.

3.2.2. Technical Levels

Additional to the methodology of related functions, a hierarchy of cybernetic control structure is established. Figure 3.11 shows a domain controller based automotive system architecture as described by Reinhardt and Kucera (2013). The task is now to deploy and integrate the logical implementation onto the depicted subsystems $E \in \mathcal{E}$.

The elements of the technical hierarchy may be parametrized to achieve a goal of energy distribution.

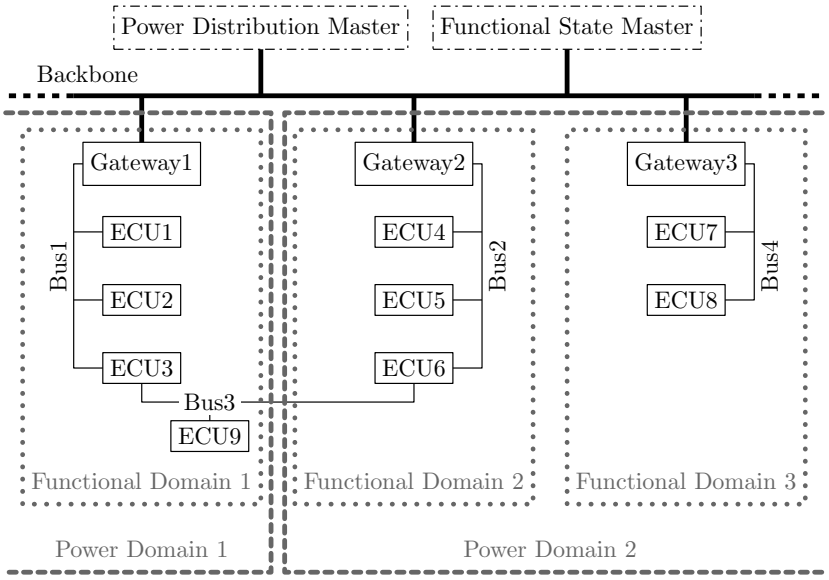


Figure 3.11.: Illustration of technical hierarchy from Barthels et al. (2012a). The vehicle is subdivided into different functional domains which are interconnected by a backbone. Different sets of subsystems form different power domains (high voltage, low voltage, ...).

3.2.3. Energy Distribution

For energy distribution, it typically suffices to change the power states of the underlying hardware in their respective plans.

The underlying hardware can be categorized as:

- Sources of electric energy, such as an alternator or a DC to DC converter. Activating a higher power state may provide additional energy flow from the source.
- Sinks for electric energy, such as heating systems. These sinks which may be throttled in order to divert the energy flow.

Figure 3.12 depicts an exemplary plan change within different power management modules. This coordinated change may effectively adjust energy flow according to the current target and strategy. Plans 1 and 4, 2 and 5, and, 3 and 6 are assumed to fulfill the requirements of States 1, 2, and 3 respectively. The different plan sets $\{\gamma_1, \gamma_2, \gamma_3\}$ and $\{\gamma_4, \gamma_5, \gamma_6\}$ may be chosen so as to support a strategic goal as part of the cybernetic control hierarchy.

Energy distribution typically refers to mid- to long-term measures. The scope of power distribution in contrast is about short-term actions.

3.2.4. Power Distribution

For ensuring safe operation in terms of voltage stability, relations in between Jobs and subsystems can be defined to impose constraints on system-wide schedules. This way,

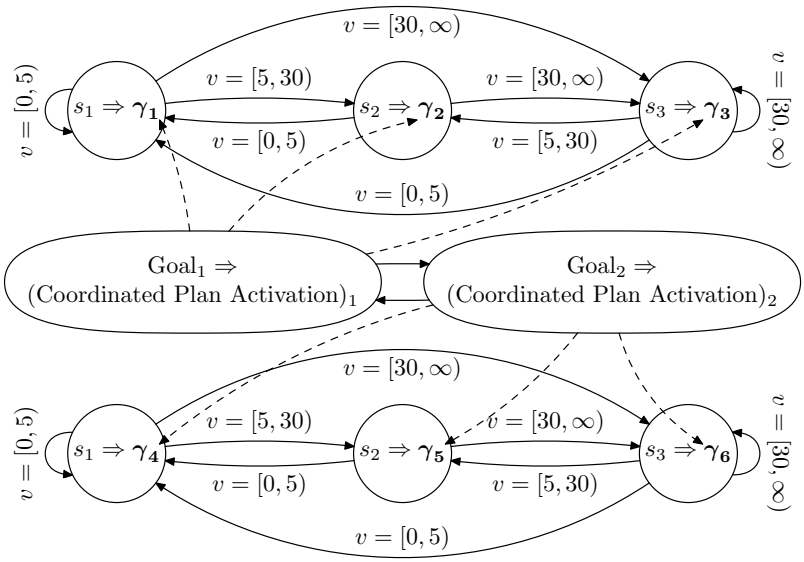


Figure 3.12.: Master Switch of State Response (Barthels et al., 2012a)

scheduling can both reflect application timing requirements as well as scheduling of instantaneous power draw and ensuring safe operation.

Definition 13 (*Stability–Criticality Relation*)
$J \rightarrow \mathrm{Crit}_{\mathrm{label}}$ with a discrete label set $\mathrm{Crit}_{\mathrm{label}} = \{\texttt{critical}, \texttt{noncritical}\}$.

Constraints may be imposed on concurrent occurrences of criticality labels. Thus, in a system wide schedule the superposition of critical jobs may be avoided. The following chapter deals with the formal methods for finding schedules regarding these constraints.

3.3. Summary

This chapter introduces the core mathematical definitions for power management in the operating system within this thesis. The concept is built around PMPs, which may be changed due to transducing system inputs. These power management modules may be affected by higher order strategy and cybernetic control schemes.

The next chapter deals with the process of modeling correctness of plans, with the target of tool support for system integration.

4. System Integration Methods

Since the logical and technical architecture of a distributed embedded system are often fixed, the question arises in what sense the integration process can be done in an energy aware way.

In order to work with the formal models defined in Chapter 3, tool support is desirable. During engineering a system based on these models, the following questions are being posed early:

- Is the current model instance correct? Such as, is the system integrable in that way? ($\rightarrow$ model checking)
- Is the incomplete model correctly completable? Is there a valid integration respecting the partial specifications? ($\rightarrow$ design space exploration)
- If so, what are possible completions? Enumerate possible ways of system integration. ($\rightarrow$ design space exploration)

In incomplete models, typically just the logical functionality, and the hardware are described. Possible questions are where to place which part of software (partitioning), and once placed, how to schedule the software.

In order to do model checking and design space exploration, suitable meta model definitions had to be crafted.

4.1. Satisfiability Meta Model

In order to reason about model and integration validity in early design phases, the mathematical models described previously have to be transformed into a description accessible to solving constraint satisfaction problems.

Figure 4.1 shows the classes involved in the process. These classes are described next along with their invariant relationships, noted as *facts*.

Abstract Classes

Figure 4.1 depicts several abstract base classes. These classes comprise timing, structured planning, and execution properties. Abstract `TimeableObjects` feature basic scheduling properties, like time at start, a required period for periodic jobs, and a duration.

Abstract `StructuredObjects` additionally relate `TimeableObjects` with hardware elements and a scheduling structure relative to `TimeableObjects`. Thus, the notion of scheduling is extended from purely time-triggered to a mixture of time and event

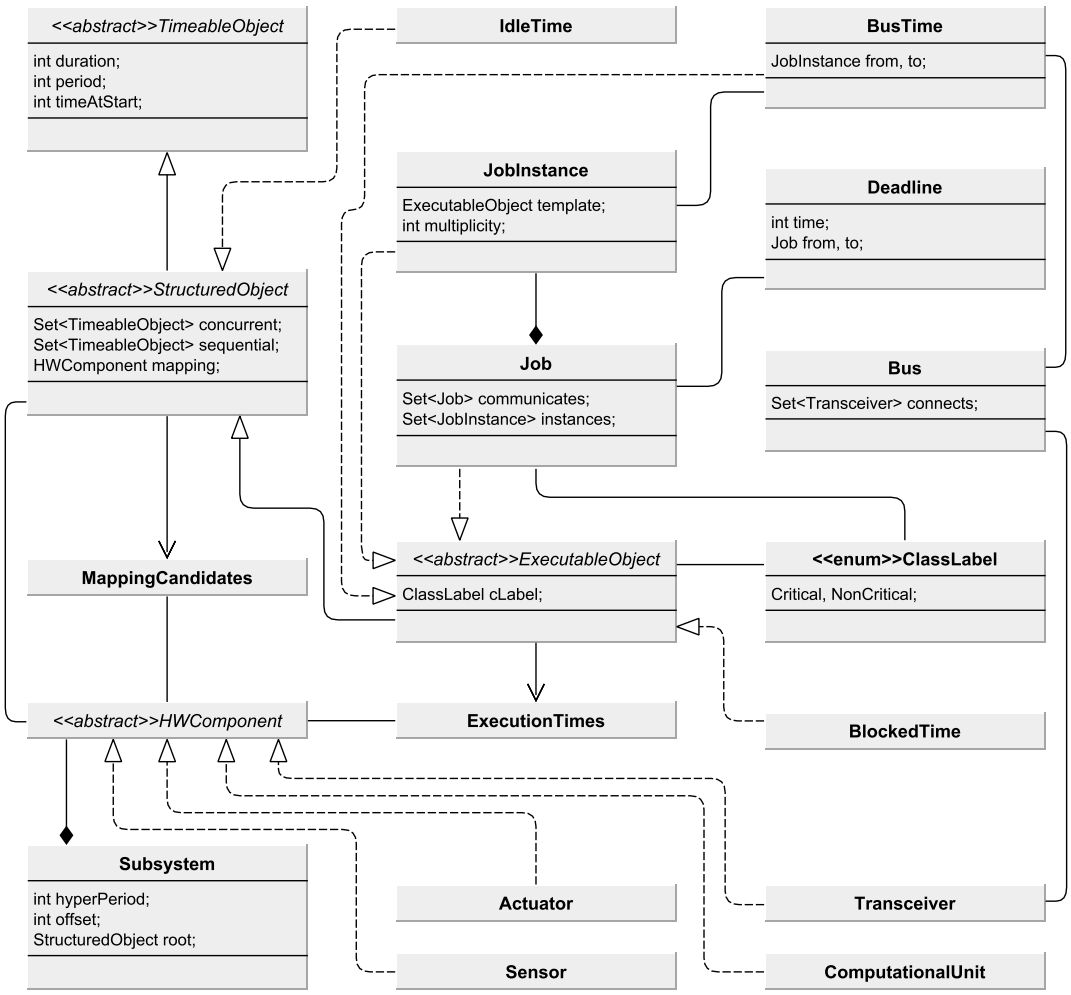


Figure 4.1.: UML Meta Model for Checking and Design Space Exploration of System Integration Variants

triggered. Events are subsumed by the start or the completion of `TimeableObjects`. This is indicated by the concurrent and sequential relationship.

`ExecutableObjects` additionally comprise execution properties. These properties are, e.g., the criticality in regard to system-wide stability, or the worst case execution time dependent on each hardware component and its power state. The class labels denote the criticality during phases of concurrency within the distributed system. They allow to define critical sections in time, where the amount of power draw is exclusive to some `ExecutableObject` among all critical ones. In that way, the problem of voltage stability in automotive power nets can be tackled.

Executable Objects

Executable objects can either be `BlockedTimes`, `Jobs`, or their `JobInstances`. These objects feature execution times depending on the actual hardware mapping. This is encoded in the `ExecutionTimes` class.

`Jobs` and their `JobInstances` utilize the subsystem hardware components to deliver distributed applications. If Jobs j with period $p(j)$ are mapped to a subsystem with a larger hyperPeriod, $p(j) < \texttt{hyperPeriod}(\mathcal{M}_F(j))$, there exists a `JobInstance` shifted by a multiple of the period $m \cdot p(j)$ for every period within the hyper period.

There is one fact which restricts the search space in a way that all $g \in \texttt{JobInstances}$ match their $\texttt{template}(g)$:

fact `JobInstanceRelation`

$$\begin{aligned}
\forall g \in \texttt{JobInstance} : \texttt{template}(g) \in \texttt{Job} \cup \texttt{BusTime},\\
\neg\exists j \in \texttt{Job} : \{\texttt{template}(g) \neq j, g \in \texttt{instances}(o)\},\\
\texttt{duration}(g) = \texttt{duration}(j),\\
\texttt{period}(g) = \texttt{period}(j),\\
\texttt{timeAtStart}(g) = \texttt{timeAtStart}(j) + \texttt{multiplicity} \cdot \texttt{period}(j),\\
\mathcal{M}_F(g) = \mathcal{M}_F(j),\\
\texttt{cLabel}(g) = \texttt{cLabel}(j).
\end{aligned}$$

Communication between subsystems is done using `Buses`. They are characterized by scheduling Bus usage as `BusTime` objects. The following fact declares that Bus usage is always related to actual and direct communication in between `Jobs` and their `JobInstances`.

fact `BusTimeIntermediary`

$$\forall b \in \texttt{BusTime} : \texttt{to}(b) \subseteq \texttt{communicates}(\texttt{from}(b))$$

As a preparatory step to check correct scheduling of end-to-end deadlines and precedence constraints, the set of affected Jobs and their BusTimes have to be tracked. The operator $\widehat{\ }$ denotes transitive closure on relations.

fact `deadlineAffectedExecutables`

$$\forall d \in D : \texttt{affectedJobs}(d) = \texttt{communicates}\hat{}(\texttt{from}(d)) \cap \left(\texttt{communicates}^{-1}\right)\hat{}(\texttt{to}(d)),$$
$$\texttt{affected}(d) = \texttt{affectedJobs}(d) \cup$$
$$\{b \in \texttt{BusTime} : \texttt{from}(b) \in \texttt{affectedJobs}(d), \texttt{to}(b) \in \texttt{affectedJobs}(d)\} \cup$$
$$\{\texttt{from}(d)\}$$

Hardware

In order to integrate all components into a runnable system, all `StructuredObjects` have to be mapped to their respective hardware. The hardware components are grouped into subsystems which are interconnected by buses.

Typically, all hardware setup is given, and the question to answer is, whether the present hardware can support a distributed application. A common approach is to incrementally integrate functions in order of their priority.

4.1.1. Discretization of Time

The standard scheduling problem is extended to generalized constraint satisfaction. To be solvable, time values need to be discretized and limited. Time is divided into a finite set of discrete slots using an affine linear transformation on the bounded limits. Also,

1. time values are required to be positive:

   fact `timePositive`

   $$\forall o \in \texttt{TimeableObjects} : \texttt{timeAtStart}(o) \geq 0, \texttt{period}(o) > 0, \texttt{duration}(o) \geq 0$$

2. the devolution of time for the sequential and concurrent relationship has to hold:

   fact `timeDevolution`

   $$\forall s \in \texttt{StructuredObject} :$$
   $$\forall s\prime \in \texttt{concurrent}(s) : \mathcal{M}_F(s) \neq \mathcal{M}_F(s'),$$
   $$\exists e \in E : \mathcal{M}_F(s) \in \texttt{components}(e), \mathcal{M}_F(s') \in \texttt{components}(e)$$
   $$\texttt{timeAtStart}(s) = \texttt{timeAtStart}(s\prime)$$
   $$\forall s' \in \texttt{sequential}(s) :$$
   $$\exists e \in E : \mathcal{M}_F(s) \in \texttt{components}(e), \mathcal{M}_F(s') \in \texttt{components}(e)$$
   $$\texttt{timeAtStart}(s') = \texttt{timeAtStart}(s) + \texttt{duration}(s)$$

Having specified the classes and universal facts which bound the search space for the partitioning and scheduling, valid solutions can be specified by defining predicates which may change their value during design space exploration.

4.2. Design Space Exploration

The aforementioned meta model can be specified in modeling languages such as Alloy (Jackson et al., 2013), or Microsoft FORMULA. These languages and tools help to explore the solution space bounded by the amount of objects and the facts restricting their relations.

Integers are present in the model as a discrete object each. Thus the scope of integers is chosen in a limited way.

The solution space is bounded by the facts. A valid solution is characterized by fulfilling additional relational constraints described as *predicates*.

4.2.1. Single Subsystem Scheduling

As part of the single subsystem scheduling, the following two predicates constraining the period windows have to hold:

1. The first predicate describes that for a solution to be valid, no `TimeableObjects` T except `JobInstances` G, $o \in T \backslash G$ may last longer than their period:

   predicate `PeriodWindow`

   $$\neg \exists o \in T \backslash G : \texttt{timeAtStart}(o) + \texttt{duration}(o) \geq \texttt{period}(o)$$

2. The second predicate states that for `ExecutableObjects` except `JobInstances`, these must exist a unique `JobInstance` for each valid multiplicity:

   predicate `periodMultiplicity`

   $$\begin{aligned} \forall j \in : \texttt{period}(j) < \texttt{hyperperiod}(\mathcal{M}_F(j)) \Rightarrow \\ \exists g_1 \in \texttt{JobInstances}, \neg \exists g_2 \in \texttt{JobInstances} : \\ g_1 \neq g_2, \texttt{busyObject}(g_1) = j, \texttt{busyObject}(g_2) = j, \\ \texttt{multiplicity}(g_1) = m = \text{multiplicity}(g_2), 0 < m < \frac{\texttt{hyperperiod}(\mathcal{M}_F(j))}{\texttt{period}(j)} \end{aligned}$$

For completing a single subsystem schedule, the following predicates have to hold additionally:

1. Resources need to be allocated unambiguously, meaning a hardware resource is occupied by at most one job at a time.

   predicate `unambiguousResourceAllocation`

   $$\begin{aligned} \forall o_1 \in T : \neg \exists o_2 \in T : o_1 \neq o_2, \mathcal{M}_F(o_1) = \mathcal{M}_F(o_2), \\ \texttt{timeAtStart}(o_1) \geq \texttt{timeAtStart}(o_2), \\ \texttt{timeAtStart}(o_1) < \texttt{timeAtStart}(o_2) + \texttt{duration}(o_2) \end{aligned}$$

2. All jobs are sequential or concurrent and at least transitively reachable, using operator $\hat{}$, from an initial root job:

   predicate `oneRoot`

$$\begin{aligned}\forall e \in E : \exists j, \mathcal{M}_F(j) = e \Rightarrow \texttt{planroot}(e) \in \mathcal{M}_F^{-1}(e), \forall j \in \mathcal{M}_F^{-1}(j) :\\ j \notin \{\texttt{concurrent}\hat{}(j) \cup \texttt{sequential}\hat{}(j)\}\\ j \neq \texttt{planroot}(e) \Rightarrow j \in \{\texttt{concurrent}\hat{}(\texttt{planroot}(e)) \cup\\ \texttt{sequential}\hat{}(\texttt{planroot}(e))\}\\ \texttt{timeAtStart}(\texttt{planroot}(e)) = 0\\ \forall j_1 \in \mathcal{M}_F(e) : \neg\exists j_2, j_3 \in \mathcal{M}_F(e) : j_1 \in \texttt{concurrent}(j_2), j_1 \in \texttt{sequential}(j_3)\end{aligned}$$

3. All deadlines between pairs of tasks hold (as defined in Definition 6 on page 35). At first, a helper function is defined, which is used later on in the actual predicate:

   fun `deadlineTimeDistance(from,to)`:Int

$$\begin{aligned}\texttt{timeAtStart}(\texttt{from}) + \texttt{duration}(\texttt{from}) \leq \texttt{timeAtStart}(\texttt{to}) \Rightarrow \{\\ \texttt{timeAtStart}(\texttt{to}) - \texttt{timeAtStart}(\texttt{from}) - \texttt{duration}(\texttt{from})\\ \}\text{else}\{\\ \texttt{timeAtStart}(\texttt{to}) - \texttt{timeAtStart}(\texttt{from}) - \texttt{duration}(\texttt{from}) + \texttt{period}(\texttt{to})\\ \}\end{aligned}$$

   predicate `deadlinesHold`

$$\begin{aligned}\forall d \in D, \forall a \in \texttt{affected}(d) :\\ \texttt{deadlineTimeDistance}(\texttt{from}(d), a) \leq \texttt{time}(d),\\ \text{let } \texttt{pre} = \Big\{\big(\texttt{communicates}^{-1}\big)\hat{}(a) \cap \texttt{affected}(d) \cup\\ \{b \in \texttt{BusTime} : b \in \texttt{affected}(d), \texttt{to}(b) = a\}\Big\},\\ \forall p \in \texttt{pre} : \texttt{deadlineTimeDistance}(\texttt{from}(d), p) + \texttt{duration}(p) \leq\\ \texttt{deadlineTimeDistance}(\texttt{from}(d), a)\end{aligned}$$

For symmetry breaking, additional facts may be used to prevent introduction of `TimeableObjects` (especially `IdleTimes` I) which are not necessary:

fact `noUnnecessaryIdleTimes`

$$\forall i \in I : \texttt{concurrent}(i) = \emptyset, \neg\exists i_2 \in I : i_2 \in \texttt{sequential}(i)$$

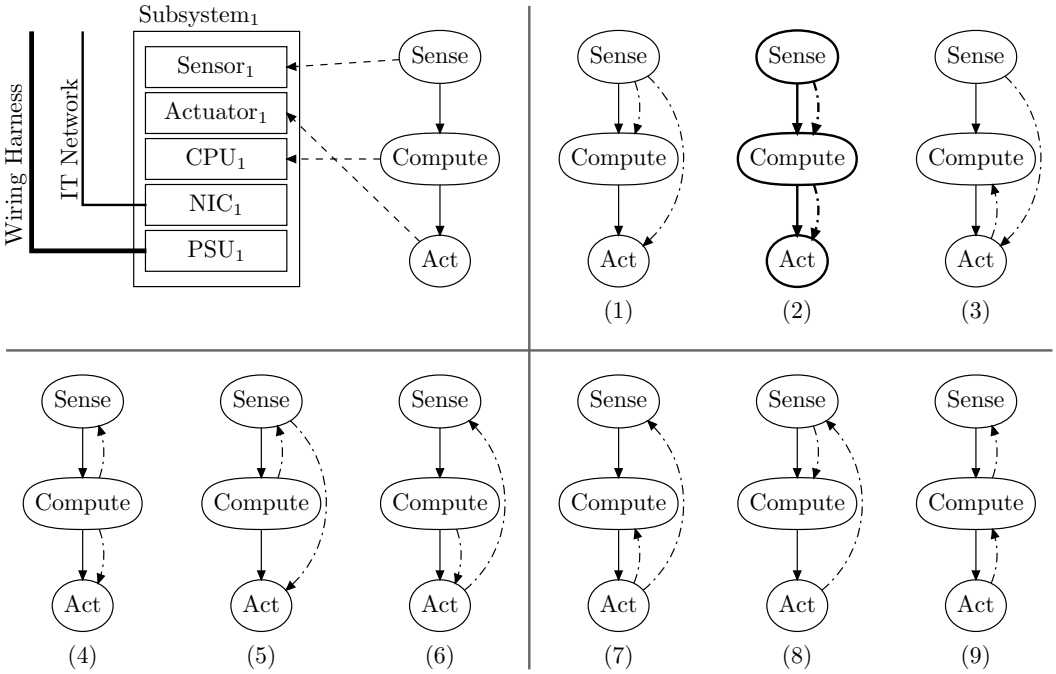


Figure 4.2.: Example of walking the solution space for scheduling within 1 subsystem $E \in \mathcal{E}$. ($\rightarrow$ Integration Level); not depicted is the special case, where all software units are scheduled concurrently.

Sequential Subsets

For prolonging idle times, subsets of jobs can be aggregated to form a sequence of tasks. Within this sequence, every job yields the hardware immediately to the following job.

The problem domain of scheduling in the structured approach introduced in the previous chapter provides many combinatorial solution candidates. Possible schedules for a small example are given in Figure 4.2. The upper left quadrant characterizes the example to be of one subsystem, and three jobs. The communication pattern is fixed being Sense $\rightarrow$ Compute $\rightarrow$ Act. The other three quadrants show the possible ways of ordering the jobs in the offline scheduling phase. Only option (2) delivers a schedule where the deadline from Sense to Act is within one period cycle interval. All other options need at least 2 periods. Depending on the deadline requirements, they still may be valid and considerable. Especially, for the case of voltage stability in the wiring harness.

4.2.2. Multiple Subsystems

When adding multiple subsystems and jobs lacking a unique mapping into the integration process, the scheduling gets more complicated. The question is also to integrate different functional chains as in the first quadrant of Figure 4.2 into one integrated schedule per subsystem.

Every job needs to be mapped to exactly one subsystem, and thus, the sets of objects mapped to each subsystem are disjunctive.

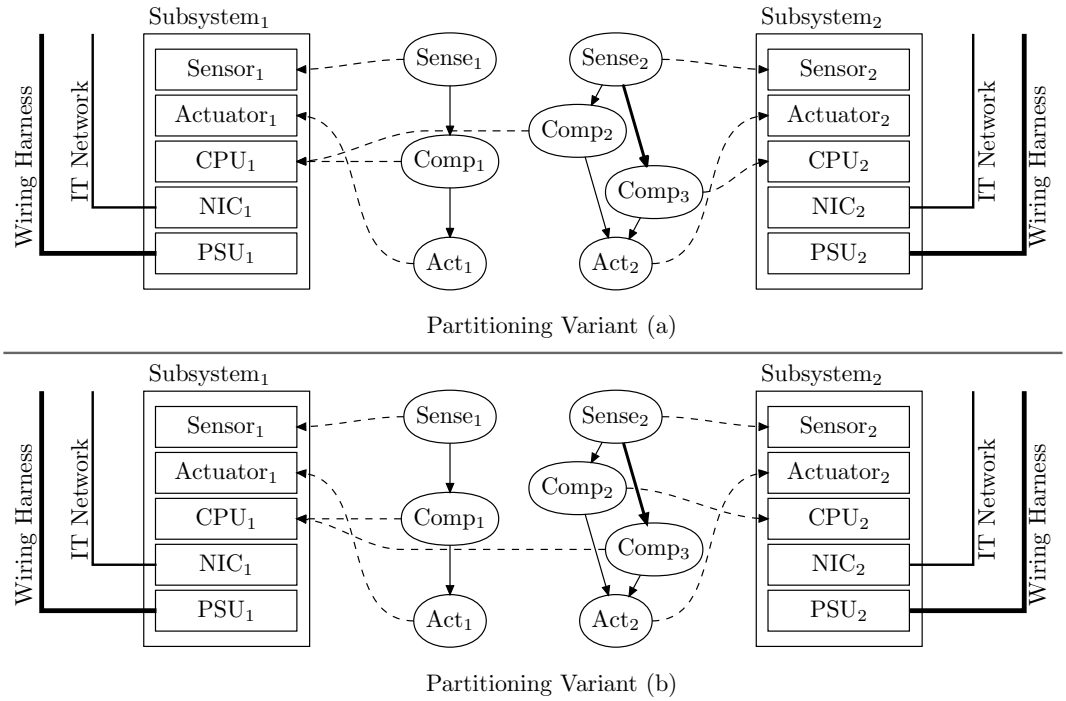


Figure 4.3.: Example Finding a Mapping and Afterwards Scheduling (→ Integration Level).

fact `allMapped`

$$\forall e_1 \in E, e_2 \in E : \qquad e_1 \neq e_2 \Rightarrow \mathcal{M}_F^{-1}(e_1) \cap \mathcal{M}_F^{-1}(e_2) = \emptyset$$

Which jobs there are to integrate, is a matter of exploring partition variants, which are highly intertwined with the problem of scheduling.
Partitioning variants mark the non-dynamic assignment of resources to deployable units. On the one hand, a partition is valid only if it is schedulable and thus integratable. On the other hand, a schedule is only valid if it respects the current partitioning and additional requirements.

In order to tackle the problem, this thesis proposes to solve both problems at the same time.

Input Dependencies

If a partitioning leads to a job needing an input with tight end-to-end deadlines being mapped to a different subsystem than the dependencies, jobs cannot be sequenced in all cases. Such a case occurs if jobs have shorter best than worst case run times.
In this case, one can resort to traditional time triggered cyclic scheduling of executable objects, or apply a mixture of time and event-triggered scheduling.

Another reason for exact time triggered scheduling is the execution of critical jobs in regard to voltage stability in the wiring harness.

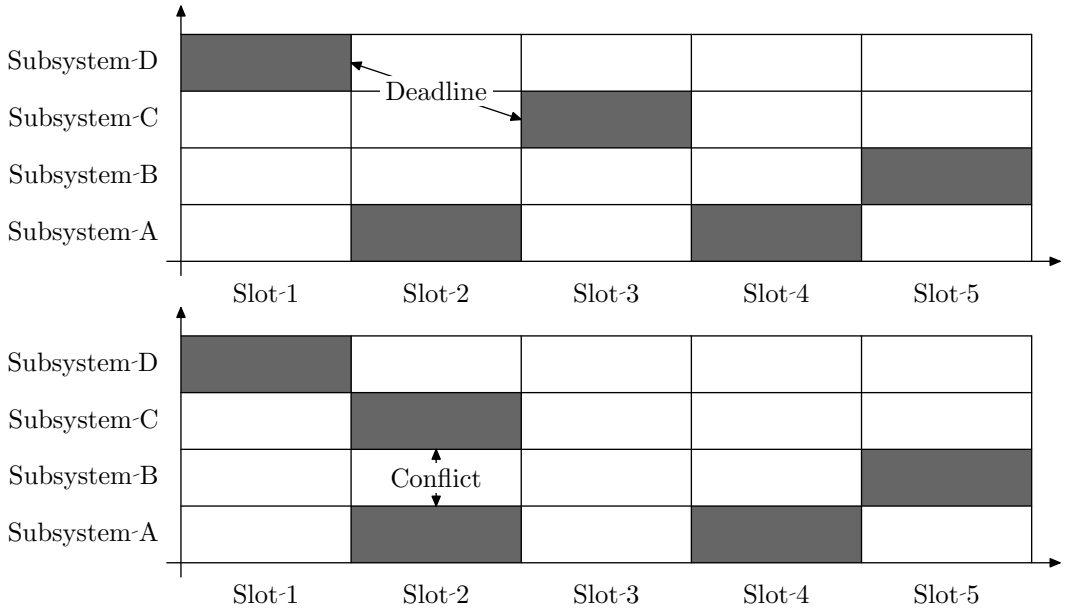


Figure 4.4.: Combinatorial Problem of Critical Power Draw

Allocating Slots for Transient Power Draw

Figure 4.4 depicts the combinatorial scheduling problem of power draw. In general, at any discrete interval in time, the instantaneous power draw must be limited to a given maximum value.

For simplicity, jobs are annotated with a label of criticality, and at most one job labeled critical can be scheduled into a column at a time. This mutual exclusion is put into effect by the following predicate:

predicate `NonConcurrentCriticalBNS`

$$\forall j_1 \in J : \texttt{label}(j_1) = \texttt{critical} \Rightarrow \\ \{\neg\exists j_2 : j_1 \neq j_2, \texttt{label}(j_2) = \texttt{critical}, T_{j_1} \cap T_{j_2} \neq \emptyset\}$$

4.3. Framework

The approach was implemented within a framework as described in Walla et al. (2013).

4.3.1. Modeling

The framework distinguishes modeling the software, hardware, integration, and simulation parts of distributed embedded systems. It was used to create the following sample case.

Job j	period(j)	communicates(j)	label(j)	mappingCandidates(j)
Sense_1	15	$\{\text{Comp}_1\}$	noncritical	$\{\text{Sensor}_1\}$
Comp_1	15	$\{\text{Act}_1\}$	noncritical	$\{\text{CPU}_1\}$
Act_1	15	$\emptyset$	critical	$\{\text{Actuator}_1\}$
Sense_2	30	$\{\text{Comp}_2,\text{Comp}_3\}$	noncritical	$\{\text{Sensor}_1\}$
Comp_2	30	$\{\text{Act}_2\}$	noncritical	$\{\text{CPU}_1,\text{CPU}_2\}$
Comp_3	30	$\{\text{Act}_2\}$	noncritical	$\{\text{CPU}_1,\text{CPU}_2\}$
Act_2	30	$\emptyset$	critical	$\{\text{Actuator}_2\}$

Table 4.1.: Software Features

	Sensor_1	CPU_1	Actuator_1	NIC_1	Sensor_2	CPU_2	Actuator_2	NIC_2
Sense_1	3	–	–	–	–	–	–	–
Comp_1	–	3	–	–	–	–	–	–
Act_1	–	–	5	–	–	–	–	–
Sense_2	–	–	–	4	3	–	–	4
Comp_2	–	4	–	2	–	3	–	2
Comp_3	–	3	–	2	–	4	–	2
Act_2	–	–	–	–	–	–	4	–

Table 4.2.: Software Worst Case Execution Times Dependent on Hardware

4.3.2. Sample Case

The sample case depicted in Figure 4.3 was translated into an Alloy specification with the properties given in Tables 4.1, 4.2, 4.3, and 4.4.

For demonstrating the capability to mix different jobs with different periods, the left chain in Figure 4.3, is assigned a period of 15, while the right chain features a period of 30. In addition, Jobs Comp_2 and Comp_3 may well be mapped to Subsystem_1. To demonstrate the feature of constraining the concurrent power draw within the system, both Act tasks are assigned the `critical` label.

Table 4.2 depicts the worst case execution times, masked out by the mapping candidates of the jobs. Mapping communicating jobs onto different subsystems will introduce a communication reservation timespan that amounts to the worst case execution time of the job assigned to the NIC.

4.3.3. Results

The aforementioned model was run once using the Minisat Prover (JNI) engine. Finding the first solution to the predicate `schedule` yields the result given in Figure 4.5. Due

deadline	from	to	time	affected
Deadline_1	Sense_1	Act_1	9	$\{\text{Comp}_1,\text{Act}_1\}$
Deadline_2	Sense_2	Act_2	16	$\{\text{Comp}_2,\text{Comp}_3,\text{Act}_2\}$

Table 4.3.: Deadlines as used within the sample case. One deadline per functional chain.

Integer	TimeableObject	Deadline	Subcomponent
6	15	2	8
Bus	**BNSClass**	**BlockedTime**	**BusTime**
1	2	0	2

Table 4.4.: Scope of Objects in Alloy Sample Case

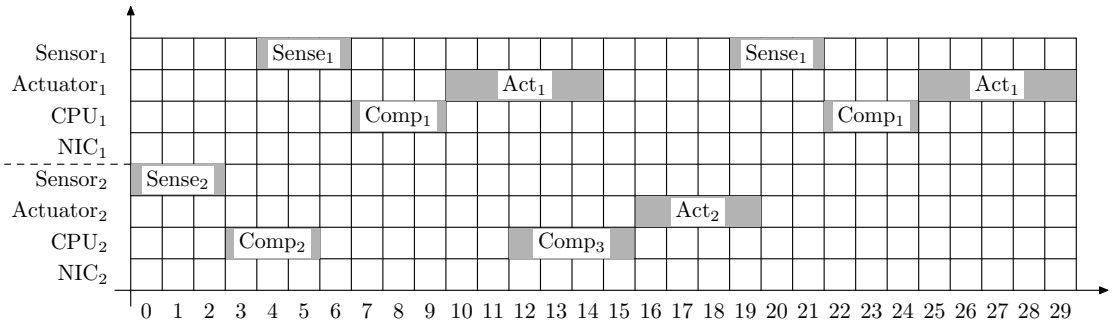


Figure 4.5.: First Result Achieved when Using the Minisat (JNI) Engine on the Sample Case

to the criticality of both Act_1 and Act_2, the first cycle schedule for both chains may not be done straight forwardly. The jobs contained in the functional chain on $Subsystem_1$ are scheduled in direct sequence of one another, while the solver shifted the execution of $Comp_3$ and Act_2 in the other chain to a later point in time.

In order to guide the search process for solutions, one may introduce additional constraints. Figure 4.6 depicts the first result, when having $Comp_3$ constrained to be mapped to $Subsystem_1$. It can be seen that the chain from $Sense_1$ to Act_1 is scheduled directly in sequence of each other. The other jobs start their execution chain with $Sense_2$ at time 22. The time for sending the sensor value over the bus is then overlapping the subsystem's hyperperiod boundary. $Comp_3$ operates on $Subsystem_1$ and sends its results back to Act_2 on $Subsystem_2$.

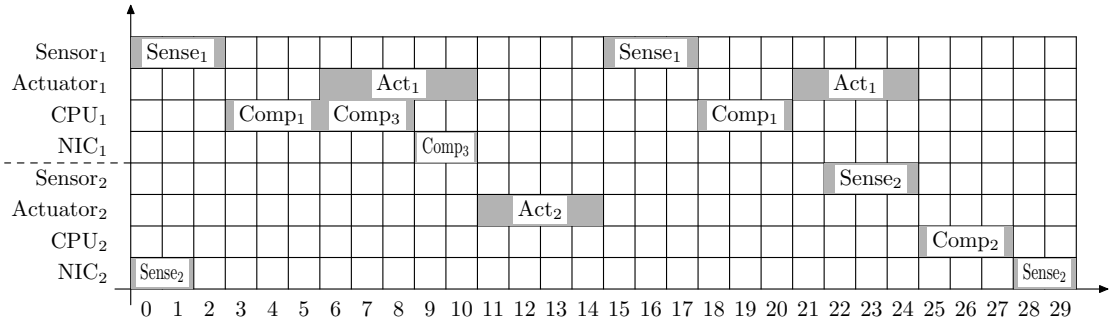


Figure 4.6.: First Result Constraining $Comp_3$ to CPU_1.

4.3.4. Simulation

Complete system specifications, as may be found by both manual or automated exploration of the design space, can later on be evaluated in regard to power and performance aspects using the simulator built into the framework. This helps to find the best out of a number of valid solutions. For this task, different visualization interfaces were integrated into the framework as part of interdisciplinary projects (Duvnjak et al., 2013; Brachert, 2013).

4.4. Summary

This chapter copes with the formal methods used for preparing partial models for system integration. For this process an existing modeling framework was extended. The framework utilizes the Alloy language and analyzer for model checking and design space exploration.

Building upon the theoretical methods and tools, a sample case with two subsystems is presented. The sample case features incomplete partitioning and scheduling specification and presence of tasks critical to voltage stability.

Building and testing a platform to actually run arbitrary Power Management Plans (PMPs) and to provide time synchronization for system-wide mutual exclusion of critical tasks is handled in the following part of the thesis.

Part II.

Implementation and Evaluation

5. Linux Implementation

This chapter explains the implementation of the platform capabilities defined in Chapter 3. It allows to distribute software over the hardware units and runs the corresponding Power Management Plans (PMPs). In order to provide the capabilities on the Linux platform, multiple OS components were modified and new ones integrated in a layered architecture. Figure 5.1 depicts the major components and their interaction to provide all functionality. The adaptivity layer just below the API is comprised of dynamically loadable kernel modules, while the capabilities at the scheduling and hardware abstraction layer are indivisibly compiled into the kernel, as it is common in Linux.

The chapter is structured along Figure 5.1. The following section first explains the configuration module and the corresponding interface using a virtual file system. The configuration is translated into data structures and loaded into the plan scheduling component. The plan scheduling component traverses the data structure, schedules tasks, and instructs the idle process as well as the tick scheduling component according to plan. For the plan to execute timely and not on arbitrary time slices, the tick scheduling is adjusted to match planned event expirations where appropriate.
The tasks which are being scheduled according to plan, communicate with each other using the communication middleware inside the operating system. The middleware mediates typed datastreams in between any combination of local or remote tasks. It also handles flushing the ringbuffer of the logging module to a central logging database. The logging is done by the logging component, which uses deferred processing to handle inputs from any context, be it hard interrupt, or user space tasks. Afterwards, the skeleton of the user space tasks is presented, as it is used during the experiment. Ultimately, the Precision Time Protocol daemon (PTPd) along with the modifications and extensions for plan scheduling are described.

5.1. Configuration and Virtual Filesystem

The virtual file system provided by the configuration module allows to create, manipulate, and destroy data structures used in the implementation.

It reads and translates configuration files and manages kernel internal data structures. The data structures are allocated and initialized, as well as destroyed depending on the configuration file.

The initialized data structures are being fed into the respective components, like plan scheduler and middleware. The plan scheduler basically needs a set of annotated graphs which can be traversed, and the communication middleware needs a set of routes which allows to flexibly interconnect software endpoints.

Besides loading data structures, the configuration can be queried by reading files within the virtual file system. This way, the status can be checked using a terminal on the subsystem as well.

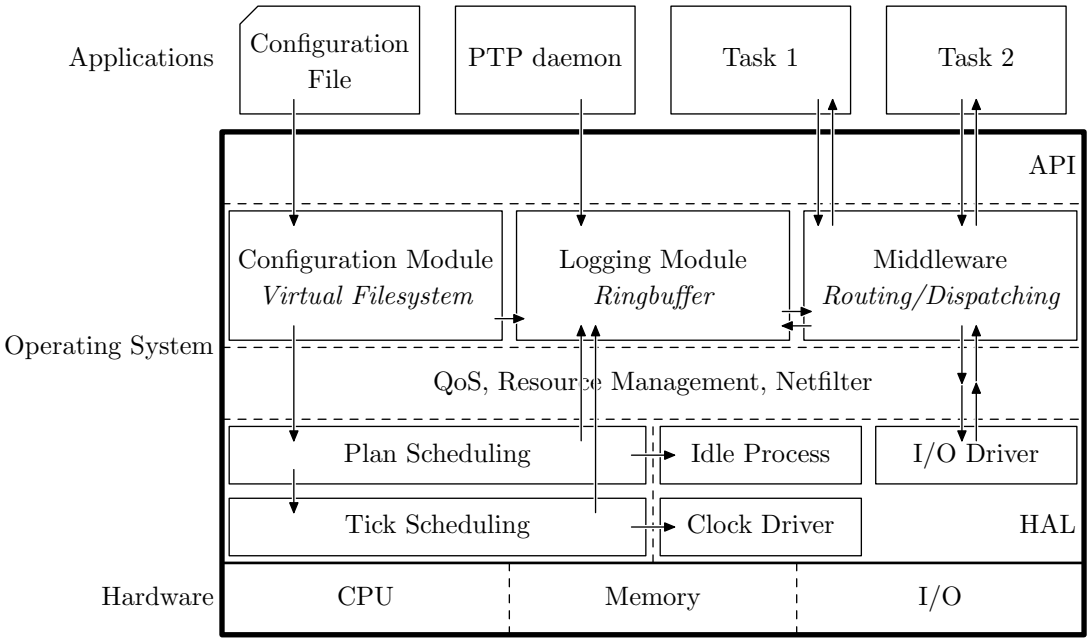


Figure 5.1.: Major Subsystem Component Interaction Overview

5.2. Plan Scheduler

The scheduling discipline for PMPs, SCHED_PM, is integrated as part of the Linux real-time scheduling class, as depicted in Figure 5.2. In this way, the plan is evaluated and traversed upon each call to the schedule function, which tests the runqueues and task states and performs dispatching of tasks to the cpu.

The plan scheduler interfaces to the logging module and traces all events related to scheduling. The scheduling can be of hardware power states, which are dispatched using the Idle Process component, and of tasks, which have to associate themselves with PMP jobs.

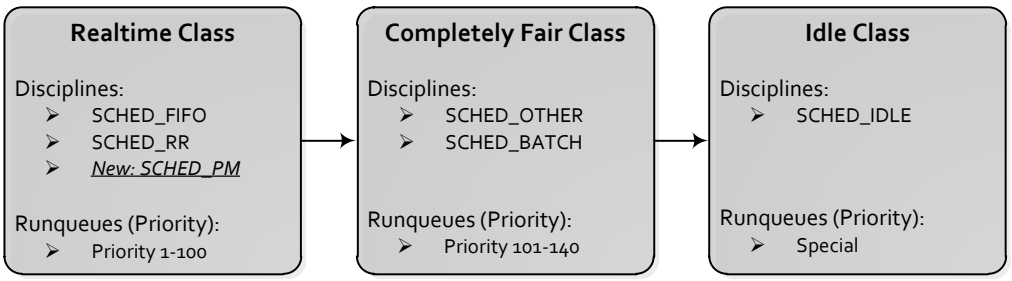


Figure 5.2.: Linux Scheduling Classes, Along with SCHED_PM Policy

5.2.1. Idle Process

The Linux idle process makes use of so-called governors for selecting and dispatching power states. The governor for NO_HZ systems (cf. Section 2.4) is called *menu*. The menu governor is used in the implementation. It features its own heuristic for selecting power states based on a sliding average on interrupt rates and the Quality of Service (QoS) requirements for latency.

The governor is adjusted in order to incorporate PMP power states. This is done by restricting the heuristic selection. If a high power state is required in the PMP, the system is held awake, even though the heuristic chooses a deep sleep state based on average interrupt rates and system task requests.

5.2.2. Task Interface

For a task to associate with the SCHED_PM discipline, a call to the POSIX API `sched_setscheduler` is needed. Along with the desired discipline, a unique job id has to be assigned which is used as an identifier in the PMPs.

Additionally to setting the discipline, a task can signal its completion to the plan scheduler using the same API. In this way, tasks can yield for synchronization to an event and be distinguished from tasks yielding because of the completion of their job. This allows both purely actor-oriented tasks to be scheduled as part of a PMP, as well as any other Linux task.

5.2.3. PMP Data Structure Association

In Linux, every CPU has a separate runqueue structure to reduce locking. The master runqueue (`struct rq`) references separate runqueues for completely fair, and real-time class processes. It was extended to include a reference to a PMP associated with the corresponding CPU in Gleixner (2011). Figure 5.3 depicts the major attributes of power management plans within the Linux kernel.

The PMP is stored using nested composition. Each PMP stores a list of nodes, which in turn store a list of edges referencing adjacent nodes.
For runnability, the plan stores a queue of events which each associate exactly one edge. An event can be any action described by the edge, i.e., the rise or fall of the edge's signal. Upon traversal of the plan, the status of nodes, edges, and the event list is constantly updated.

During run-time, the timing definitions are followed as well as the logical combination of events using logic operators.

5.2.4. Sequential Logic Operators

The implementation of processing sequential logic in the kernel does not do optimization of the logical formula at hand. This way, the performance of the implementation can be tested using randomly generated, tautological plans of different sizes. All delays are being put to zero, so just the computation and propagation of signal levels is assessed. The plans are generated by recursively and randomly applying the following substitution

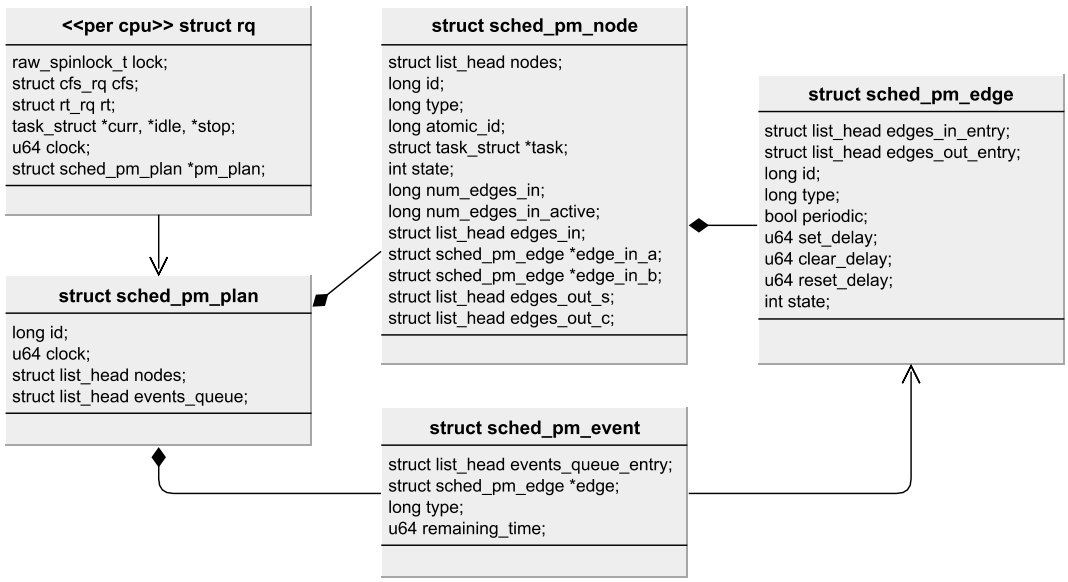


Figure 5.3.: Power Management Plan Data Structure as Held Within the Scheduler

rules:

$$\mathrm{id} = \mathrm{id} \circ \mathrm{id}, \quad \neg = \mathrm{id} \circ \neg, \quad \mathrm{id} = \mathrm{id} \wedge \mathrm{id}, \quad \neg = \neg \vee \neg.$$

The first step in the recursion starts with the identity function id. Since some of the rules rewrite id and ¬ by a function over two signal inputs, the plan size rises exponentially. Thus, the repetitive application of these substitution rules yield large plans, of which the execution time can be rated.

5.3. Tick Scheduler

For a CPU resource to be shared fairly and evenly among tasks, a time division multiplex scheme is used. Typically, the time is sliced into uniform slices using system tick events. The Linux system tick triggers a call to the schedule routine, which operates on the runqueue data structures, schedules, dispatches, and potentially preempts processes.

The Linux tick scheduling subsystem is adjusted to match the event sequence dictated by the current PMP. The tick scheduling component actively forwards, stops, and restarts the timer associated with the system tick. This is done by parameterizing the clock hardware using an associated clock driver.

The plan setup is to determine the accuracy of scheduling cyclic tasks together with power states, so as to provide adaptive quality of service over time. Figure 5.4 shows a cyclic plan, scheduling jobs with bringing the CPU up to a high power state featuring lesser latency before the actual start of the job.

The goal is to verify the suitability of the plan scheduling mechanism to periodic tasks with strong timing requirements. Again, all events occurring within the tick scheduling

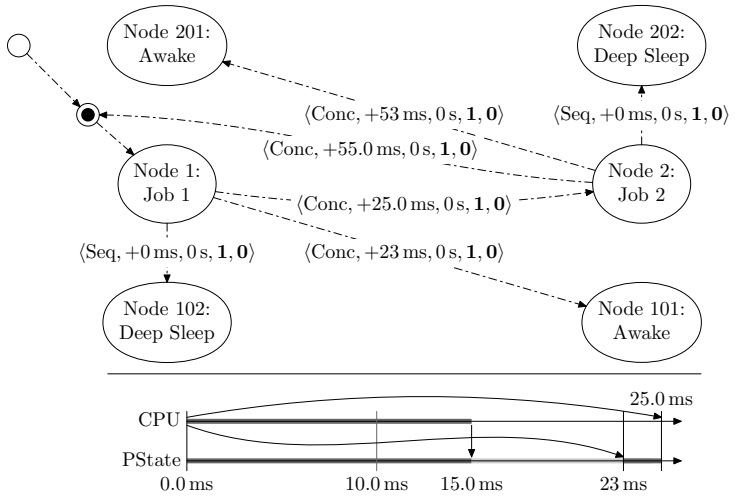


Figure 5.4.: Functional Chain Set for Validation of Deep Power States

framework are traced using the logging module. Two dummy tasks were utilized for the evaluation.

5.3.1. Dummy RT Tasks

There are two identical real-time tasks in the experiment, denoted Job 1 and Job 2 in Fig. 5.4. Both are started upon the experiment start and terminate after the predefined duration of the experiment. Algorithm 1 outlines the program code. Upon start-up, they register themselves with the scheduler, specifying the novel SCHED_PM discipline. The priority during the test run was chosen for both tasks to be 20. Meaning they will be inserted into queue with priority 20 upon the adaptive tick which is aligned to their starting time.

Algorithm 1: Real-Time Task Program

```
input: ExperimentDuration, TaskID, Priority
/* At first, register with the scheduler:                          */
sched_setscheduler(ProcessID, SCHED_PM, TaskID, Priority);
/* Track experiment start:                                         */
StartTime ← gettimeofday;
while gettimeofday() < StartTime + ExperimentDuration do
    LoopStart ← gettimeofday();
    /* Randomly choose Task duration in interval [1, 20] ms:       */
    LoopDuration ←(rand() mod 19) + 1;
    while gettimeofday() < LoopStart + LoopDuration do
        /* nothing                                                 */
    end
    yield();
end
```

After each time the tasks are being scheduled, they spend most of their time calling

`gettimeofday()`. When scheduled, they take a random time within $[1, 20]$ ms to complete. Since this is a Linux system function, this amounts to a large portion of time in kernel space.

The task signals the completion of its job to the scheduler using a special call in the end. This call is different from the default `sched_yield()`, which may still be used and allows default non actor-oriented Linux tasks to be part of a PMP as well.

5.4. Multicasting Middleware

The middleware for multicasting typed streams of information between one to many communication partners was implemented as part of different theses and interdisciplinary projects (Fuchs, 2012, 2013; Totakura, 2012).

The distributed system has multiple stream endpoints within the application layer. These applications interact with the operating system using network sockets. The sockets are both used for sending and receiving network data as well as for configuring the broadcast manager present in the Linux kernel.

The network performance is largely limited by the underlying hardware. An analysis of this is out of the scope of this thesis.

5.4.1. Socket Interface

The Linux kernel features a low-level framework for Controller Area Network (CAN) bus systems together with a set of high-level socket family extensions since Linux version 2.6.25 (Hartkopp et al., 2013). Usage of these socket family extensions is patented for the case of vehicular functions (Hartkopp and Thürmann, 2005).

Every task can open a socket and instruct the middleware to filter and dispatch incoming messages as well as to enqueue outgoing messages for sending. When handling a packet originating from an arbitrary interface or local socket, a routing table is considered, specifying zero to many target rules for a data stream.
In this way, different local tasks can communicate with one another, as well as communication can be replicated (multi-casted), onto external networking interfaces.

Quality of Service

Enqueuing messages for external interfaces passes the QoS framework of the Linux kernel. This framework allows different streams of data to be prioritized and accounted for. This works especially well for Ethernet and Internet links.
For embedded system buses used in automation or the automotive domain, the prioritization is often handled in hardware. One example for this is the CAN bus. For this type of bus, kernel work queue based multiplexing of data streams is applied (Fuchs, 2012).

Netfilter Hooks

Linux provides different filtering methods for incoming traffic. These methods are used to retrieve data for the communication middleware. For CAN buses, the low-level

CAN framework features so-called receive lists with CAN ids and socket endpoints for userspace tasks.
For Ethernet, e.g., the so-called netfilter framework can be used for a similar task. Upon receiving of an Ethernet frame, the middleware can use its routing table for dispatching and replicating of the message content. In this way, content can be routed from Ethernet to CAN and vice versa.

5.4.2. Transducing Machines

Additionally to mediating between endpoints, the middleware features a rule set for transducing notifications to actions regarding changes of PMPs. This implementation was part of an interdisciplinary project (Fuchs, 2012).

δ				ω
Rule Active	From State	Input	To State	Plan Response
1	1	0	2	1
0	2	1	1	0
...				

Table 5.1.: Encoding of Transducing Machines as Lookup Tables in the Communication Middleware

Table 5.1 depicts the tabular specification of a transducing machine (cf. Chapter 3), as implemented in the middleware. Each state has a set of rules which get activated and deactivated respectively. The plan scheduling module is notified asynchronously of the requested plan change. The plans get changed upon reaching an end node.

Changing plans triggers the start node of the next plan upon entering the end node of the current plan. Each plan has it's own clock value. This clock value is inherited by the new plan. Thus the correctness in switching plans and inheriting timings can be tested by utilizing two plans as in Figure 5.4. The first plan does not allow as deep sleep states as the second plan. One may change in between these plans while maintaining seamlessly the schedule of the jobs.

The tick scheduling experiment incorporates such a plan change and is explained in more detail in the following chapter.

5.5. Logging Subsystem

The logging subsystem features a preallocated ring buffer. Figure 5.5 depicts the data structure and data fields used within the logging entries. Since the logging module is part of a layered architecture, it can handle arbitrary payloads. A call to log an entry first needs an 8-bit logging source identifier. The logging subsystem then attaches the source identifier and a timestamp value to the payload.

The ring buffer has to be flushed explicitly as part of a PMP node. This way, logging data may be flushed explicitly after an experiment run, or at well defined points within. For the tick scheduling and logic operator tests, this flushing happens after the experiment run. For tests which take a long time, it might be necessary to flush this buffer

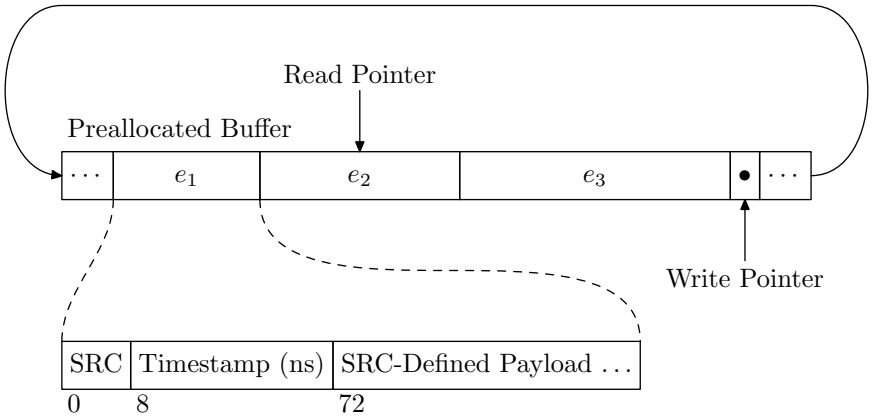


Figure 5.5.: Ring Buffer and Data Format Used in Logging Module

during runtime. This was e.g. the case with the test described in the next section.

5.6. Precision Time Protocol daemon

In order to be able to provide implicit and explicit support for voltage stability mechanisms, a means of time synchronization was evaluated in regards to the model presented in Chapter 3. The basis for the analysis is the PTPd as developed by Correll et al. (2005). It is a software only implementation of the IEEE1588 protocol as defined in Chapter 2. Due to the software only nature, it features a control loop which is designed to cope with significant levels of input noise.

5.6.1. Implemented Control Loop

The PTPd makes use of filtering and of a PI controller for adjusting the Frequency Locked Loop (FLL) in the NTP subsystem of the Linux kernel (Mills et al., 2010). This original control system is depicted in the upper part of Figure 5.6. Due to the interface to the FLL subsystem, the clock is slewed not only for adjusting clock speeds, but also to compensate for clock offsets in general.

Assume to be given a periodic job with cycle time 500 ms. If the offset is 500 ms, a whole cycle in kernel time will be left out or be added in the process of eliminating the clock offset.
In order to reduce this systematic error, one might want to resort to hard setting of the clock. This approach has the disadvantage that time may virtually get lost and applications malfunction. The gradual compensation for offsets is more robust in terms of application compatibility.

As a contribution of this thesis, an additional plan control loop is implemented in the PTPd. The corresponding system diagram can be found in the lower part of Figure 5.6. Denote |SW| the width of a synchronization window as planned in the PTPd slave system. Control the plan execution, so as to align the time of SYNC message reception to the middle of the window. This is done by introducing another PI controller mechanism which signals the true synchronization period to the plan scheduler. This is necessary,

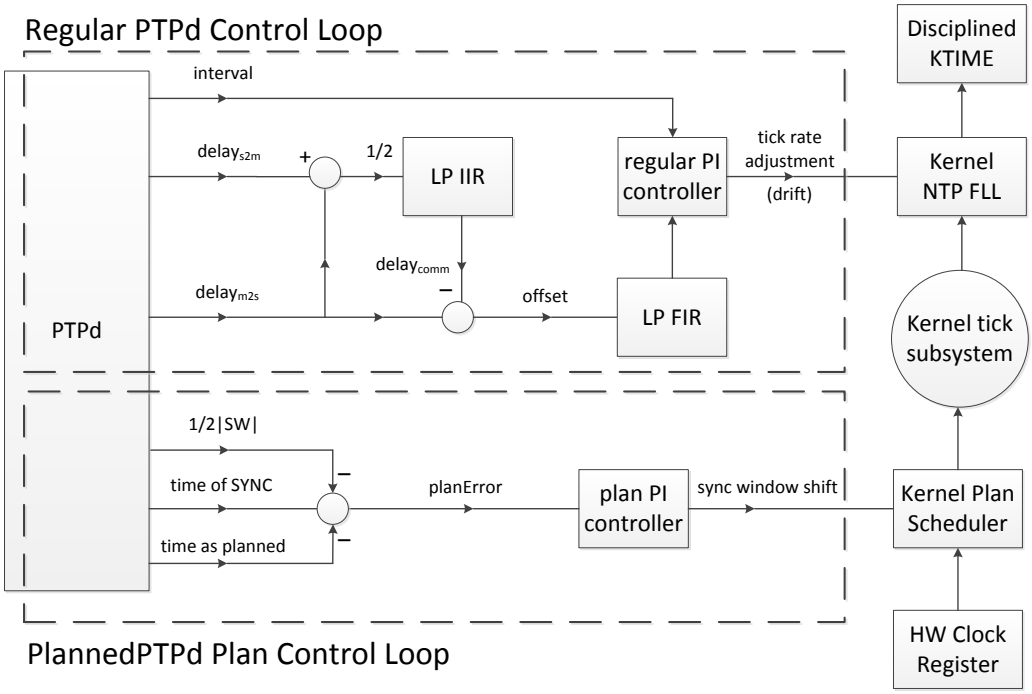


Figure 5.6.: Regular PTPd control loop as illustrated in Correll et al. (2005) together with extensions for PlannedPTPd version. To the right is the layered architecture of the scheduling, tick, and timing subsystem of the Linux kernel.

because the plan execution is not directly affected by the timing subsystem. The plan scheduler uses the monotonic runqueue clock which is directly read from the clock register and thus agnostic to adjustments.
For this scheme to work, the synchronization window size has to be chosen large enough and the communication delay within this window is assumed to observe low jitter. If jitter is introduced, another LP filter module would have to be introduced into the loop.

5.6.2. Precision Time Experiment Setup

In order to achieve great precision using Precision Time Protocol (PTP), predictability and hardware support is vital. In software-only implementations, such as in Linux PTPd, filtering and control theory is applied to yield convergence.

As an exemplary use case for power management planning, PTP convergence is benchmarked across planned and unplanned implementations in Linux under presence of high priority real-time tasks introducing jitter in protocol handling.

Implementation Extension

For the experiment setup, PTPd version 2.2.2 is extended to incorporate planning capabilities. Algorithm 2 sketches the adjustments made to the PTPd main thread.

Algorithm 2: PTP Daemon Planning Extensions to Main Thread

```
void protocol() ;                                              /* PTP HANDLE */

    sched_setscheduler();        /* register PTPd main thread as PTP HANDLE job */
    LoopStart ← gettimeofday()
    while true do
        /* Receive network packets and handle the PTP protocol in the default PTPd
        call dostate().  The following extensions and modifications apply;
        - the call does not block longer than the following condition to yield to
        the plan scheduler;
        - planError is calculated using the time of reception of SYNC.          */
        dostate();
        if gettimeofday() >= LoopStart + Deadline then
            yield();
            LoopStart ← gettimeofday();
        end
    end
```

The following modifications are made to support the planning scheduler:

- The PTPd main thread is registered as the PTP HANDLE job in the plan scheduler.
- The PTPd internal `dostate()` call is modified to respect the deadline of PTP HANDLE, and to record the `steerError` upon reception of a SYNC packet.

Additionally to the main thread, another thread used for signaling the SYNC window to the plan scheduler is introduced and sketched in Algorithm 3.
The following steps are executed:

- The new thread is registered as the PTP PI EVENT job.
- It computes and signals the $t_{\text{completion}}$ time in a PI manner using planError as input.

Algorithm 3: PTPd Planning Extensions to PI Event Thread

```
void steerThread() ;                                           /* PTP PI EVENT */

    sched_setscheduler();        /* register current thread as PTP PI EVENT job */
    while true do
        /* Calculate t_completion in a PI manner using planError as input.     */
        t_completion ← PI(planError);
        usleep(t_completion);
        /* Yield to the plan scheduler.                                        */
        yield();
    end
```

The same implementation is used for both the slave as well as the master operation modes.

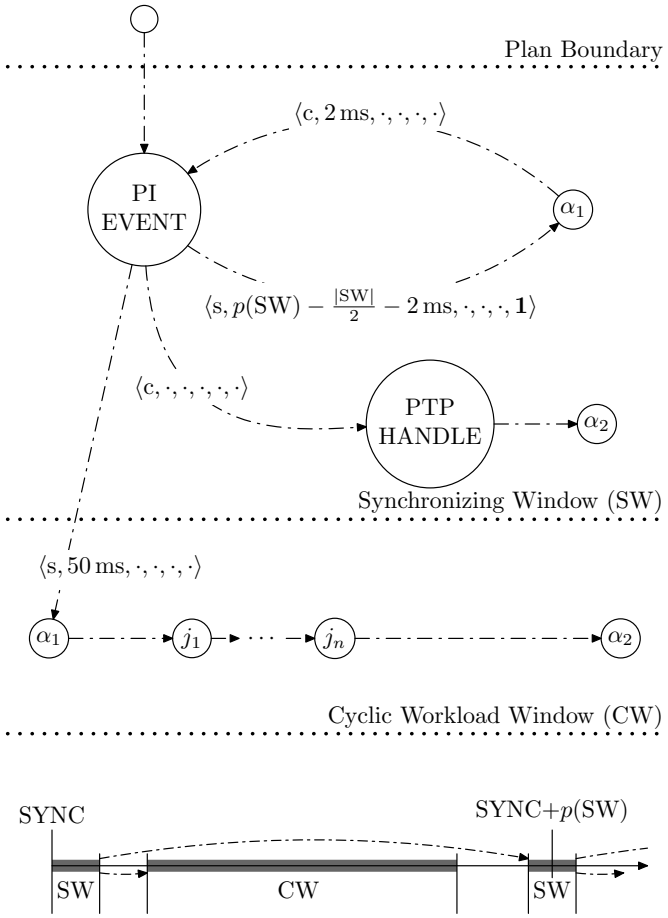


Figure 5.7.: PlannedPTPd—Plan for Slave Nodes with Synchronizing and Cyclic Windows

Slave Operation

The corresponding plan for associated slave operation is depicted in Figure 5.7. The plan allows the PTP HANDLE task to discipline the local clock, but also to adjust and shift the synchronization and cyclic workload phases of the local power management plan. This is done by shifting the end-time appropriately. A maximum of half a synchronization window length may be shifted in either direction of past and future.
This is done so as to adjust the window to the SYNC and FOLLOW_UP messages, of which the receive time accuracy is of utmost importance. The lower part of Figure 5.7 depicts the structural dependencies of this mixture of time and event-triggered windowing in an unrolled version. The power state α_1 is assumed to adhere highest quality of service, while α_2 maximizes energy savings.

Master Operation

The master operation allows the installation of different cyclic windows. The period of the synchronization window has to match the periods of the slaves. Figure 5.8 de-

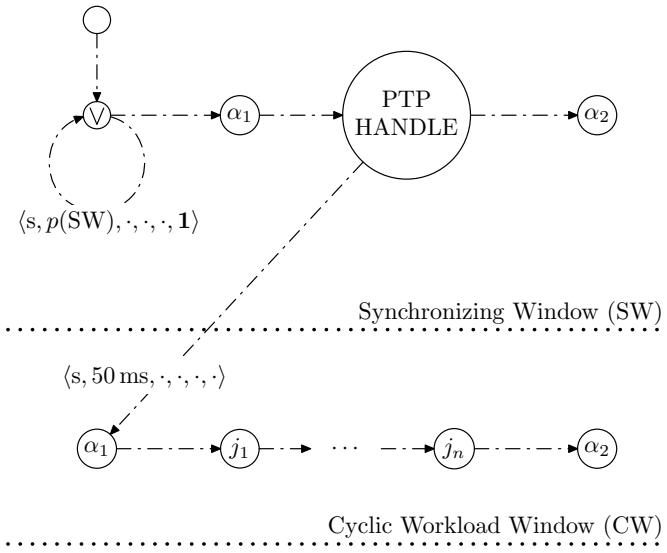


Figure 5.8.: PlannedPTPd—Plan for Master Node with Synchronizing and Cyclic Windows

picts the PMP used for the master role in the PTPd experiment. The synchronization window is characterized by running the PTP HANDLE job. It is activated at the beginning, sending out protocol data and waiting for requests during the whole window time. Besides the PTP and synchronization window, the window for cyclic workload is constructed similarly to the slave operation mode.

For master operation, there is no need to introduce special cycle signaling, because the clock is not modified.

Maximum Slewing

If the clock offset of the slave is more than 512 μs, the clock is slewed at its maximum rate, which is 512 ppm per default.

Consider the slave clock to be a lot behind the master clock. The slave's clock frequency will be sped up by the maximum rate of 512 ppm. The plan scheduling will not be affected, because it works directly with the hardware clock register. In order to align the plan to the SYNC packets, the slave HANDLE needs to be triggered sequentially of a synchronization event. The event is steering the plan in a PI way. Defining t_2 in accordance with Section 2.4.1, as the time of reception of the SYNC packet measured in the time base of the slave, the plan needs to be disciplined so as

$$(\mathrm{planError})_i := (t_2)_i - \texttt{timeAtStart}(\texttt{HANDLE})_i - \frac{|\mathrm{SW}|}{2}$$

converges to 0:

$$(\mathrm{planError})_i \longrightarrow 0, \qquad \text{as } i \longrightarrow \infty$$

In this way, subsequent steering of the plan execution has the maximum degree of freedom. If the plan is to be sped up, the PI EVENT needs to be signaled earlier. This

can be, e.g. because of slewing of the slave clock, but also in order to take a proportional step towards the right activation time.

The width of the synchronization window |SW| and thus the deadline of the PTPd has to be chosen large enough, so as the probability of

$$Pr\left\{\text{abs}(\text{planError}) < \frac{|\text{SW}|}{2}\right\} \tag{5.1}$$

is sufficiently high. This is because, without hardware support, t_2 may only be correctly determined, if the packet comes in while the PTP HANDLE job is active. The HANDLE job is active exactly within the synchronization window time span.

Reasoning for Introducing a PI EVENT

It is necessary to decouple a cycle event from the PTP HANDLE job into the plan scheduler because of the following reasons:

- Assume the absolute planError is within the bounds

  $$\text{abs}(\text{planError}) < \frac{|\text{SW}|}{2}.$$

  Choose the completion time $t_{\text{completion}}$ of PI EVENT in a proportional integral manner:

  $$(t_{\text{completion}})_i = a_P \cdot (\text{planError})_i + a_I \cdot \sum_{k=0}^{i} (\text{planError})_k + \frac{|\text{SW}|}{2}$$

  Assume a single optimistic correction step $a_I = 0, a_P = 1$:

  $$\begin{aligned}\texttt{timeAtStart}(\texttt{HANDLE})_{i+1} &\approx \texttt{timeAtStart}(\texttt{HANDLE})_i + \text{planError} + \frac{|\text{SW}|}{2} + \\ &\quad p(\text{SW}) - \frac{|\text{SW}|}{2} - 2\,\text{ms} + 2\,\text{ms} \\ &\approx (t_2)_i - \frac{|\text{SW}|}{2} + p(\text{SW})\end{aligned}$$

  Assume $(t_2)_{i+1} = (t_2)_i + p(\text{SW})$:

  $$\approx (t_2)_{i+1} - \frac{|\text{SW}|}{2}$$

  It follows

  $$\begin{aligned}(\text{planError})_{i+1} &= (t_2)_{i+1} - \texttt{timeAtStart}(\texttt{HANDLE})_{i+1} - \frac{|\text{SW}|}{2} \\ &= (t_2)_{i+1} - (t_2)_{i+1} + \frac{|\text{SW}|}{2} - \frac{|\text{SW}|}{2} = 0\end{aligned}$$

  - Let $-\frac{|\mathrm{SW}|}{2} < (\mathrm{planError})_i < 0$. If $(t_{\mathrm{completion}})_i > (\mathrm{planError})_i + \frac{|SW|}{2}$, HANDLE may finish later and thus steer the plan without an extra EVENT node.
  - Conversely, let $(t_{\mathrm{completion}})_i < (\mathrm{planError})_i + \frac{|SW|}{2}$ It is not possible to finish earlier, because the packet reception time can not be anticipated. The cycle offset shift has to be decoupled from PTP HANDLE, introducing PI EVENT. Finishing earlier may also harm s2m measurements and other PTP duties.

- Conversely, assume the planError is out of bounds. This is either a singular event, or the predictability of the system and the synchronizing window size are mismatching. The implementation drops packets if offsets and time values are too large.

Depending on the cyclical window tasks at hand, it is better to step the clock for large offsets. In general it should be stepped, if convergence is wanted within a given time frame. Assume no drift/skew term, convergence and offset being measured in the same unit:

$$t_{\mathrm{convergence}} \approx \frac{|\mathrm{offset}|}{|\texttt{PTPSlewFactor}|}$$

5.7. Summary

This chapter presents the implementation of the model defined in the first part of the thesis. Additionally, test cases for individual subcomponents are presented. These test cases involve the computation of logic operators within PMPs, the timing and tick scheduling of PMPs, and a planned variant of the PTPd.

The following chapter first presents the hardware platform used for testing the implementation and then gives the results for each test case.

6. Evaluation in a Test Bench

The scheduling concept of Power Management Plans (PMPs) presented in this thesis was implemented and evaluated in a test bench located at the institute for energy conversion technology at the Technische Universität München (TUM). The test bench features an original BMW 7 series chassis and wiring harness for investigating voltage stability dynamics. Twenty embedded systems are integrated and networked within the test bench. Figure 6.1 depicts the complete system setup along with control and measurement computers.

The evaluation of the integration of a scheduling mechanism for PMPs starts with a small scale, single subsystem setup, and continues with a distributed experiment.

6.1. Test Bench

In order to test and evaluate the implementation of the models defined in Chapter 3, a test bench resembling a typical modern automotive system was used.

Figure 6.1 shows the actual setup. The chassis along with the embedded systems in the middle as well as the computers, measurement and instrumentation hardware in the foreground. To the right is the Linux PC used for booting, providing an NFS root for the subsystems, and for gathering the logging data. To the left is additional equipment for controlling and visualizing the experiments at the same time.

6.1.1. Hardware and Network Architecture

Figure 6.2 depicts the logical architecture as setup in the test bench. The Ethernet is configured using three virtual LANs for distinguishing and shaping traffic with different purposes.

Depicted in Subfigure 6.2a is a domain controlled architecture as described by Reinhardt and Kucera (2013). The test bench consists of three domains entitled vehicle dynamics, entertainment, and car body. Each domain consists of subsystems interconnected by a Controller Area Network (CAN) bus. The domains are connected using an Ethernet backbone. VLAN 810 is used for the experiment data in a fully switched network. This experiment data can be exchanged point to point from labview with any Electronic Control Unit (ECU), or be multicasted among ECUs, where only the domain controllers can directly communicate over Ethernet with each other.

Subfigure 6.2b shows VLANs 0 and 820. VLAN 0 is used for booting the ECUs over the network, for providing a Network File System (NFS), and for logging. VLAN 820 was not used during the following experimentation. It may be used for an optional live view on the LabVIEW workstation.

The subsystems within the test bench are all of the same prototypical type.

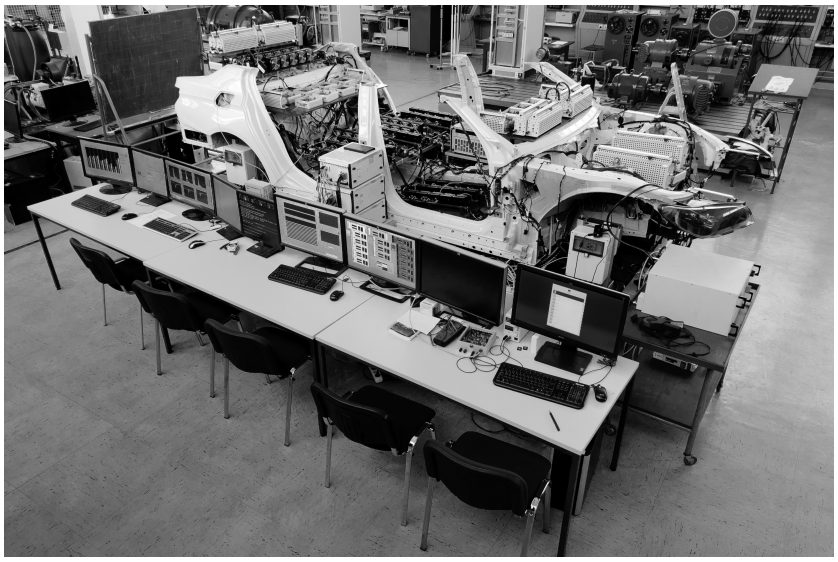
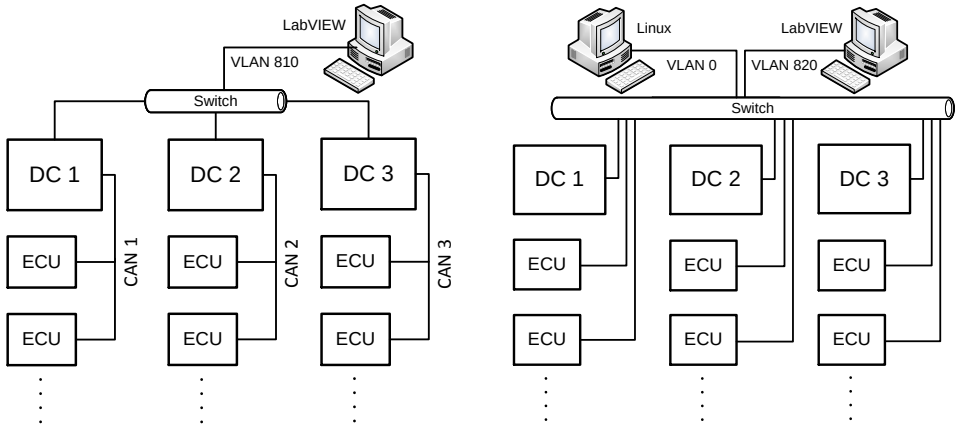


Figure 6.1.: Test Bench Setup, Picture Taken from Ruf et al. (2013a)



(a) (b)

Figure 6.2.: Distributed system topology setup (Ruf et al., 2013a). Virtual LAN 0 is used for the experimentation in this thesis. Virtual LANs 810 and 820 feature live subsystem statistics in conjunction with LabVIEW.

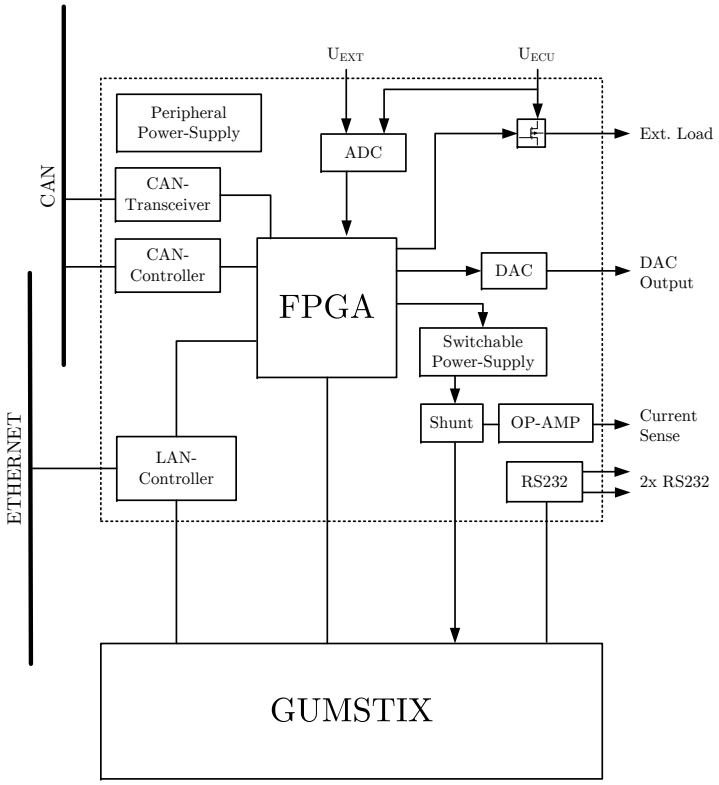


Figure 6.3.: Schematic of the Custom Platform Including the GUMSTIX Computer-On-Module (COM). The Figure is taken from Barthels et al. (2012c).

6.1.2. ECU Hardware Platform

In collaboration with the institute for integrated systems and the institute for energy conversion technology, a custom hardware platform was developed by Schlenk (2012). The design and features were published in Barthels et al. (2012c). The platform is targeted at evaluating the interaction of partitioning, efficiency, and voltage stability in automotive systems.

Custom Circuit Board

The prototypical ECU platform is based on a custom circuit board, as depicted in Fig. 6.3. The board hosts a power supply, peripheral connectivity, and the COM.

As peripherals, the boards can power dynamic loads, used to emulate sensors and actuators. For connectivity, both CAN and 100BaseTx Ethernet interfaces are provided. The CAN is being driven by an MCP2515 chip, while the Ethernet is driven by a SMSC9291.

The COM used during the experiments is an OMAP3503 based GUMSTIX Overo model, which is described next.

Power State	ARM MPU	ARM Core
C1	WFI	ON
C2	WFI	inactive
C3	CSWR	inactive
C4	OFF	inactive
C5	OSWR	OSWR
C6	OFF	OSWR
C7	OFF	OFF

Table 6.1.: Power States of the OMAP3503 MPU, which is based on the ARM Cortex A8. Taken from Linaro (2012) and Barthels et al. (2012c).

GUMSTIX Overo

The Overo series manufactured by GUMSTIX, Inc. (2012) features an ARM Cortex A8 MPU offering a multitude of operation points and idle states. The idle states used in the Linux kernel description of the hardware can be found in Table 6.1.

The following explanation of the power states was taken from a White Paper by Texas Instruments (2012) and a previous publication (Barthels et al., 2012c).

- ON: All circuits are fully operational.
- WFI (Wait for Interrupt): Equals a halt command. The logic is fully powered, but not actively working.
- CSWR (Closed Switch Retention): The logic is in retention mode, but still powered
- OSWR (Open Switch Retention): The logic is switched off, but the state is preserved in dedicated static RAM.
- OFF: The component is completely powered off

The typical power consumption of the OMAP MPU was measured using a multimeter and an oscilloscope. For the typical consumption, the system was put under load using the GNU basic calculator (Nelson, 2012) and measured in C1 (halt) under different frequencies and supply voltage levels.

Figure 6.4 depicts the measured average consumption. The higher plane is spanned by the load figures, while the lower plane shows the C1 figures. It can be seen that a significant base consumption is present regardless of the frequency or supply voltage scaling. Because of this, switching off different hardware components yields much greater savings than scaling on this platform. Thus, scaling frequencies and voltages is not used in the experiments.

FPGA

The FPGA is incorporated for running functional chains with the highest timing requirements. It is able to process inputs autonomously, like monitoring supply voltage levels and adjusting the power supply of attached actuators.

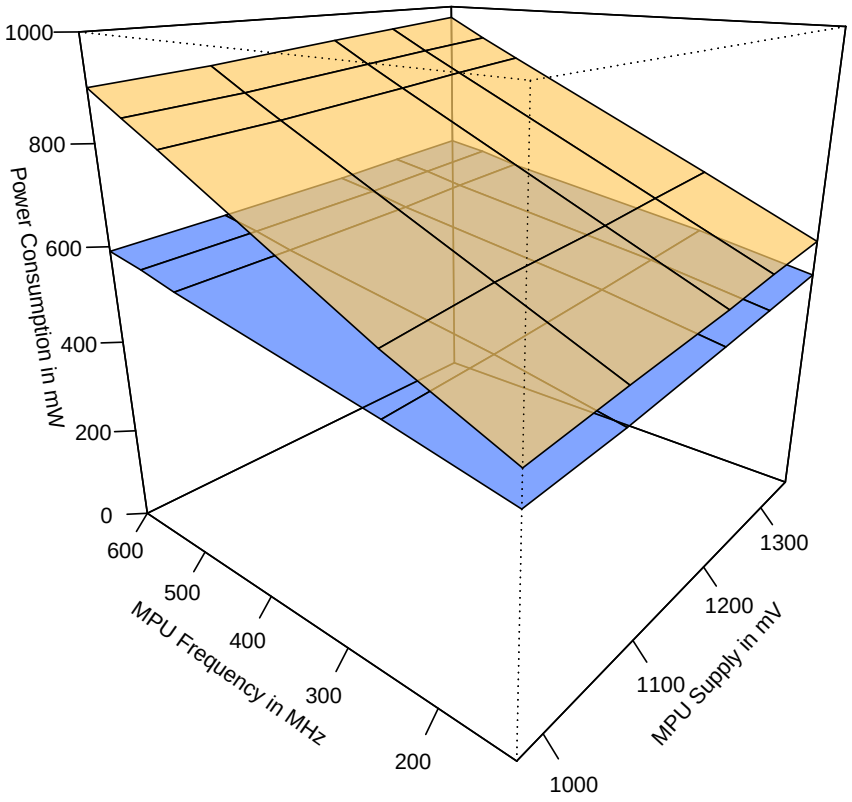


Figure 6.4.: Typical OMAP3503 MPU Power Consumption as Measured in Barthels et al. (2012c)

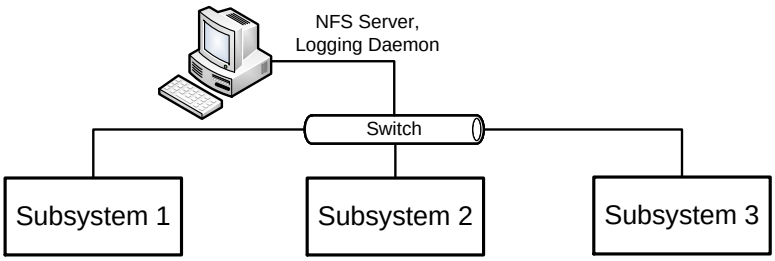


Figure 6.5.: Experiment Setup, Subsystems Connected via Ethernet

Experiment	Subsystem 1	Subsystem 2	Subsystem 3
Plan Timing	—	In Use	—
Logic Operator	—	In Use	—
PTPd	PTP Slave	PTP Slave	PTP Master

Table 6.2.: Experiment Deployment

The FPGA is not used during the following experimentation but can be used in the future for off loading additional tasks.

6.1.3. Small Scale Experimentation

A small scale setup was used in order to evaluate the implementation presented in Chapter 5. Figure 6.5 depicts the hardware topology used during the experiments. The subsystems mount their root filesystem and boot via an NFS share. The NFS share is served by a Linux PC which was also used for collecting data and analyzing the system.

Table 6.2 details the deployment of experiments to subsystems. In the first experiment used for analyzing the plan scheduler and transducing mechanism, only Subsystem 2 was used. The software setup is explained in detail in Sections 5.3 and 5.4. An important goal of this experiment is to show significant power savings without significantly deteriorating Quality of Service.

The second experiment works on the same subsystem and is used for evaluating the plan scheduler in regards to sequential logic operator performance. The corresponding implementation is described in Section 5.2. The logic operator performance builds around the rare case of using the scheduler for solving logical formulas as fast as possible. The experiment serves as a worst-case scenario since normally only very few operators are evaluated at the same time.

The final experiment is about performing time synchronization in a distributed system. The description of the PTP daemon and the extensions for power management planning are described in Section 5.6. The core of the experiment is to provide a proof of concept for the platform built around PMPs in regards to mutually excluding critical tasks in a time-division multiple access manner.

6.2. Plan Timing Experiment

For validating the implementation, firstly a single subsystem setup was chosen and different aspects of the scheduler assessed.

The implementation of the plan scheduling is purely done in software, as described in Chapter 5.

On the embedded device, an experiment script is run via the serial console, which synchronizes with the experiment control and starts the experiment and data collection. This experiment script divides the experiment interval of 60 s into three subintervals.

6.2.1. Experiment Script

In the first interval in between 0 and 20 s, the GNU basic calculator (Nelson, 2012) is used to compute thousands of digits of $\frac{\pi}{4}$ in the background. The second and third intervals from 20 to 40 and from 40 to 60 s feature no background tasks but strongly fragmented phases of busy and idle times. In the second interval, no sleep states deeper than C1 are allowed, while the third interval adds C2 and C3 (cf. Table 6.1), since the additional energy savings of deeper idle states on this platform are insignificant using the official device drivers (Barthels et al., 2012c). The transition from the second to the third interval involves the switching of PMPs.

Within all three subintervals, the system is put under the same irq load by repetitively sending Ethernet frames.

The timing of the experiment script is shown in Figure 6.6. It can be seen that immediately after the start of the script, TCPDump (2012) and the real-time processes are started. There are two TCPDump instances, one of which is instructed to receive as many packets as are generated during the experiment, while the other just filters for the experiment start packet and returns. After TCPDump returns, the PMP scheduler is activated. The real-time processes thus start to run and the system load monitoring using sysstat begins. Sysstat is collecting values on a per second basis (Godard, 2012). To induce CPU load on the system, the GNU basic calculator is used to compute 1750 digits of

$$\tan^{-1}(1) = \frac{\pi}{4}.$$

The calculation is started 1 second after receiving the start packet. The calculation gets timed and the result stored. After bc finishes, the system is mostly busy executing the real-time tasks. Because of the random task completion times as shown in Algorithm 1 on page 65, the lengths of the idle intervals and thus also the heuristic sleep state selections within Linux are expected to vary over time.

6.2.2. Experiment Control Network Sequence

During the experiment, the controlling PC generates network packets for testing the scheduler's accuracy during heavy interrupt load. The network communication sequence is depicted in Figure 6.7. The network packets are generated using an open-source tool called ANETTEST (2012), which allows to generate packets according to script files. The packets have length 107 bytes on wire. The protocol is UDP/IP with a small textual payload of 65 bytes.

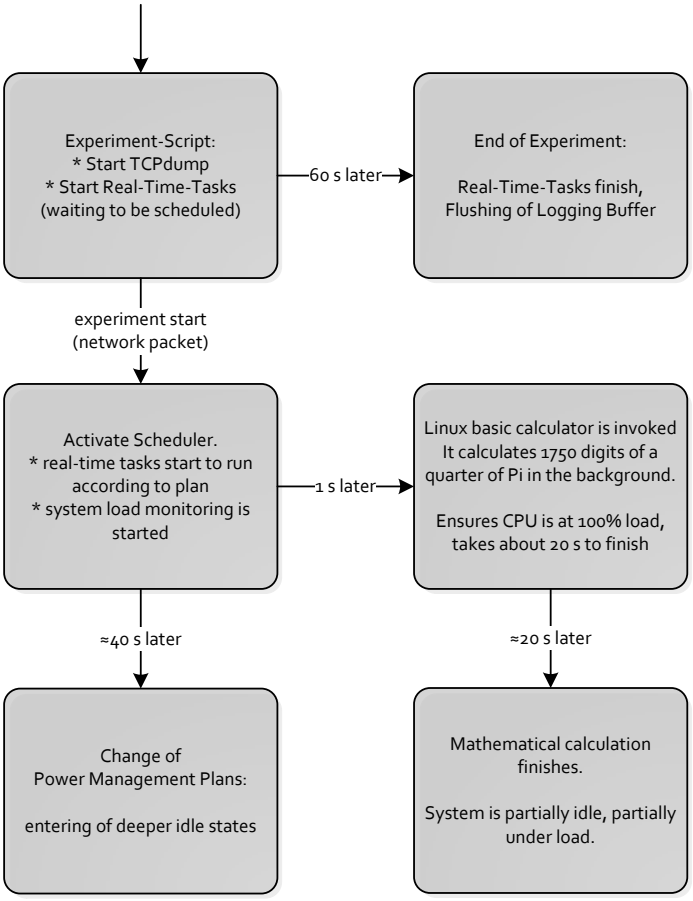


Figure 6.6.: Script as Used in the Plan Timing Experiment

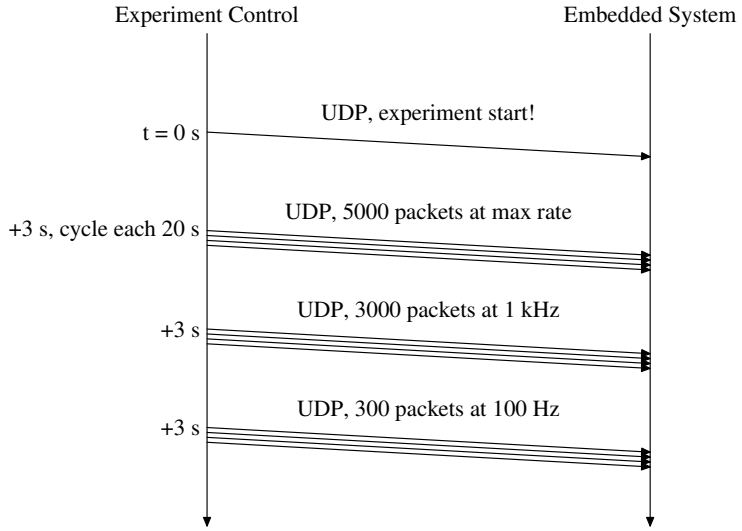


Figure 6.7.: Generated Network Packet Flow During Load Testing

Each experiment is generating one trace for kernel and scheduler events. Traced events include timed tick-scheduling, clock hardware programming, power management plan traversal and idle power state selections as computed by the Linux menu heuristic. Beside the scheduler, all network traffic belonging to the experiment is dumped to a RAM disk by TCPDump. Finally, the system load figures are being put out by sysstat. All data files are collected after the experiment via SCP.

6.2.3. Results

The power consumption results are gathered along with the system utilization in Figure 6.8. The topmost graph depicts the idle state selection computed by the Linux menu heuristic. This selection was dynamically overridden if the Awake node specified in the PMPs is active. Thus, a low latency for the actual activation of a time critical job is ensured.
The second graph shows the packet rate as it was dumped by TCPDump. A total number of 24,900 packets were sent to the embedded system. During the experiment, 619 packets were dropped by the kernel (2.5 %) and none were dropped by the interface. The packet loss occurred mostly during the phase of partial idle after second 40. It is highly likely, that the packet loss is due to a lot of packets coming in although a deeper idle state was entered. Due to the repetitive C-state selections and wake-ups, a lot of time and processing power gets lost. On network systems inherently having real-time characteristics, one might as well introduce time spans preventing deep sleep states during burst intervals into the local power management plans before hand.
The third graph shows a stacked bar chart of the CPU load over time. It can be seen, that the tasks do not produce a lot of time in user space. Most time is spent in kernel-space, because of the API `gettimeofday`. Within the first 20 seconds, the calculation of $\frac{\pi}{4}$ is done in the nice portion. Upon every burst of network packets, high soft-irq loads can be observed.

The fourth graph shows the power consumption along the previously explained experiment procedure. It is clearly visible, that all three intervals of the experiment reveal a different picture of power consumption. In the second interval, the deepest allowed sleep state is C1. The overall consumption is reduced, while two peaks are visible during phases of high network load. The third interval then adds C1-C3 states and a higher dynamic range of power consumption over time. This section continues with discussing the system latencies during the execution of the PMP depicted in Figure 5.4 on page 65.

The three graphs on the bottom of the page show cycle intervals. It can be seen that while the system is allowed to go into idle, the crucial transitions in between nodes $1 \rightarrow 2$, $2 \rightarrow 1$, and thus $1 \rightarrow 2 \rightarrow 1$ are remarkably better compared to the transition $1 \rightarrow 101$. This is due to disabling idle states well before crucial transitions.
The transition $1 \rightarrow 101$ at the end of an idle interval has higher variations in timing, because of additional wake up delay imposed by the idle states which may have been entered before and because the clock device is programmed from the interruptible idle task context.

At around second 40, the PMPs are switched to enable deeper hardware idle states. It can be seen that the switch is seamless and the timebase of each plan is transferred correctly.

The gray horizontal lines mark the mean value of the respective graph. These mean values for the crucial transitions are quite exactly according to plan. The maximum and minimum latencies are not visualized here since latency and its improvements are handled mostly by the incorporation of the RT-Preempt patch into the Linux kernel.

6.3. Logic Operator Performance Experiment

The logic operator performance was measured by stopping the time needed to execute a given number of operators, as described in Section 5.2.4. The plot in Figure 6.9 shows the results in a log-log scale. It is important to note that the scaling is highly dependent on the system architecture.

For small signal count, the plans and tracing data still fit into the cache of the processor, thus the scaling is almost linearly.
As soon as the power management plan exceeds the cache size, the implementation scales worse. For the evaluation, all the signals are computed as fast as possible, without propagation delays or preemption. Finishing a plan may only happen if there is no active signal anymore. At first, all nodes are activated and later-on deactivated. Since many different signals and operators have to be considered and reconsidered, they effectively produce an increasing number of cache misses.

Having such strongly connected combinations of logic operators with all timing constants being 0 is a rare circumstance. Typically, the signals to start a job are few and timed. If such a high operator count is needed for a type of application, a realization using a dedicated cache, or even directly in hardware has to be considered. Since PMPs resemble electronic circuitry, the transition to hardware can easily be made.

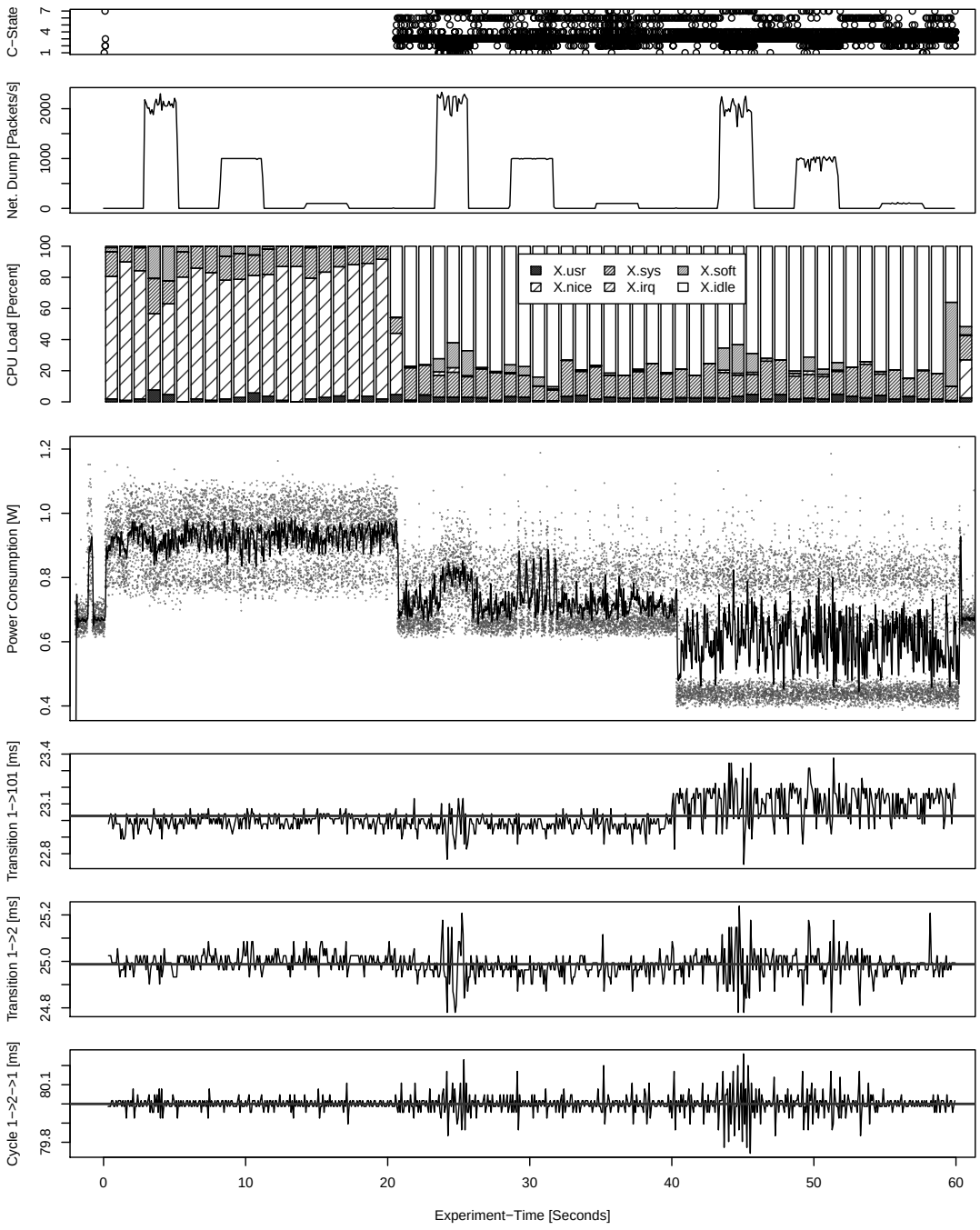


Figure 6.8.: Embedded system power consumption with varying process and interrupt load over time. The solid power consumption line is resulted after filtering using a Butterworth low-pass with cut-off frequency of 66.7 Hz.

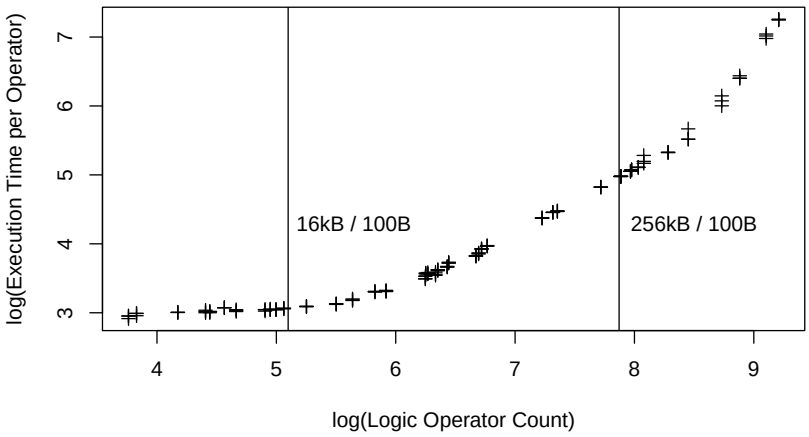


Figure 6.9.: Results from synthetic randomized sequential logic operator test as conducted as part of Gleixner (2013). Logarithmic execution time in μs over logarithmic signal and operator count. Vertical lines depict theoretical optimistic L1 data cache and L2 cache boundaries of the underlying platform.

6.4. PTPd Experiment

The experiment for time synchronization using the IEEE1588 PTP is conducted on three subsystems. Subsystem 1 and 2 are preconfigured to be slaves of the PTP master clock on Subsystem 3. All subsystems run ntpdate to step the clock at boot time. This helps to start out with offsets of less than a second. Because of running ntpdate, the subsystems are expected to start the experiment with varying offsets in their time base.

Table 6.3 presents the relevant configuration settings, as explained in Section 5.6 on page 68. The settings are applied to the following test cases:

1. The first test runs a default PTPd on a subsystem which is idle except for flushing

Long Name	Short Name	Value
All PTPd Variants		
SYNC interval	—	500 ms
Discard Offset Values	—	>500 ms
Background RT Task Priority	—	20
Planned PTPd Only		
Cycle Time	p(SW)	500 ms
Synchronization Window Width	|SW|	40 ms
Proportional Control Factor	a_P	1/2
Integral Control Factor	a_I	1/9
PlannedPTPd RT Priority	—	1

Table 6.3.: PTPd Experiment Configuration Settings

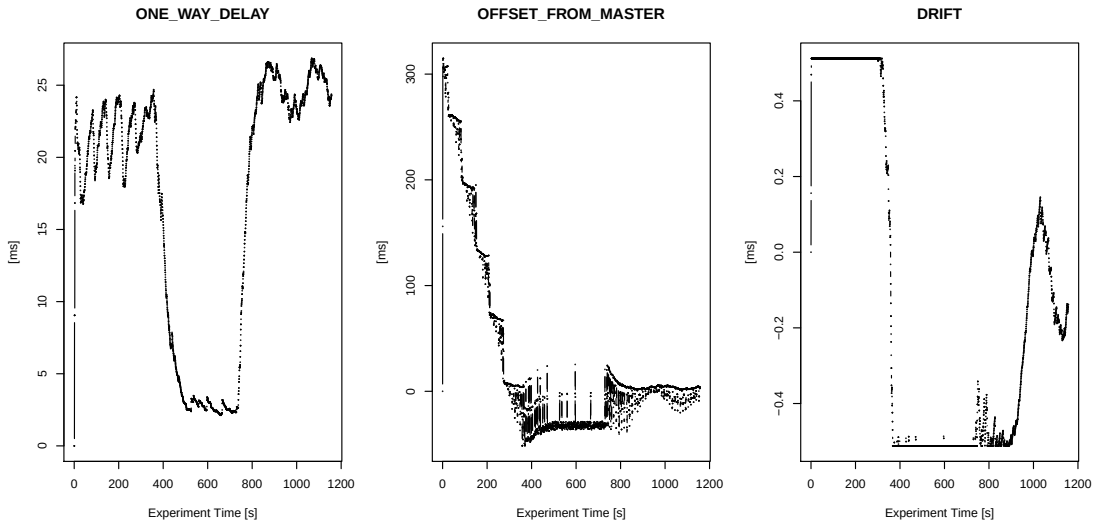


Figure 6.10.: Test Case 1; regular PTPd control system values for Subsystem 1. Subsystem is idle except for cyclically flushing these values to the Linux PC.

logging values. In this test, only SYNC interval 500 ms and the instruction to discard offset values >500 ms is used.

2. The second test adds real-time tasks with priority 20. These tasks may preempt the PTPd occasionally.

3. The third test then evaluates a PTPd version built around the PMPs defined in Chapter 5 in this thesis. The window width |SW| is chosen as 40 ms. For the PTP thread to be usable in a PMP, it is scheduled with RT priority 1. This priority level is lower than the priority level of the other real-time tasks.

   The additional control loop needs a proportional and integral factor chosen as $\frac{1}{2}$ and $\frac{1}{9}$ respectively.

6.4.1. PTPd on Idle Subsystem and Network

Subsystem 1 was idle after boot. The deepness of sleep states was unrestricted and selected according to the Linux menu heuristic. The present plan only checks and flushes the ring buffer used for logging every 100 ms. Despite this process, different Linux tasks and device drivers may preempt PTPd at any point in time. Since the system is booted using an NFS mounted root file system, network traffic is non-negligible and present as a source of indeterminism.

Since the original PTP daemon control loop features low-pass filtering and PI control, it is expected to achieve synchronization on this type of setup pretty well.
Figure 6.10 depicts the estimated and the resulting drift values as computed by the PTPd control loop detailed in Figure 5.6 on page 69. The experiment starts with an estimated offset of 300 ms, resulting in a positive drift value as the input to the kernel Frequency Locked Loop (FLL). The FLL then reduces the clock speed by 512 ppm,

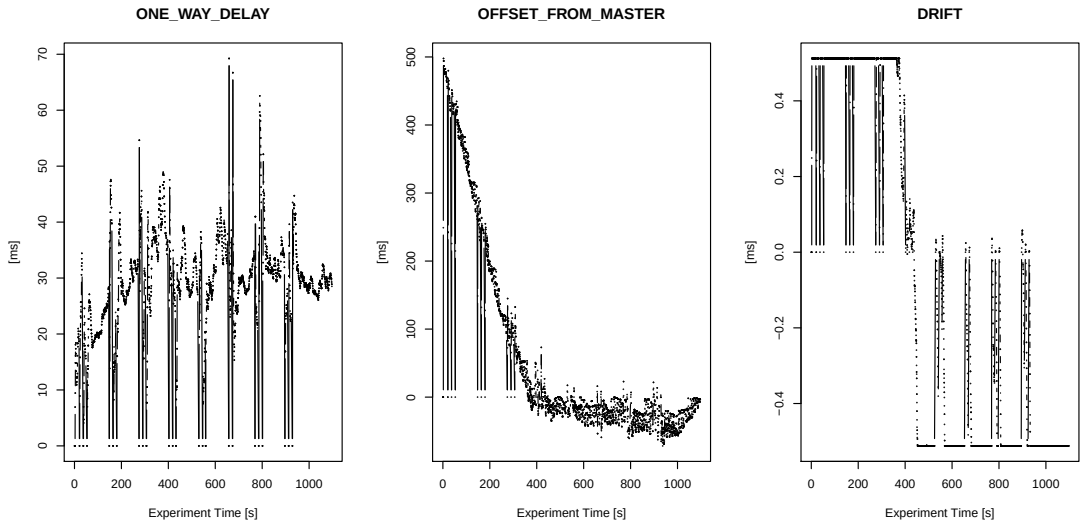


Figure 6.11.: Test Case 2; regular PTPd control system values for Subsystem 1 under presence of real time tasks. Measured values are discarded if they are off by more than 500 ms. Thus some values are intermittently clamped to zero. Due to the added noise, the time for convergence is significant.

in order to compensate for the offset. Once the offset is eliminated, the drift value is adjusted in the other direction in order to stop slewing the clock. It can be observed that the estimated values still contain some significant noise term, but the offset starts to converge to zero.

Additionally to random delays introduced by sporadic system task arrival or hardware wake-up delays, the question arises how well the daemon handles the presence of hard real time tasks.

6.4.2. PTPd with Presence of Real-Time Tasks

Additional to the setup from the previous subexperiment, cyclical hard real-time tasks are introduced. The cycle time is the same as the PTPd synchronization interval. The hard real-time schedule is dispatched according to the hardware clock register. This may not seem critical on first glance, but in the process the tasks are gradually shifted along the time line. Upon start up of the PTPd, the first occurrence of the synchronization and the real-time interval are uncoordinated. Due to the shift of the schedule versus the synchronization window, the execution of the PTP daemon will almost surely be contending the real-time tasks from time to time.
Due to the priority setup in the experiment, the real-time tasks do preempt the PTPd. This creates significant levels of noise in the control system variables as shown Figure 6.11.

If the tasks are critical for system stability, the schedule has to be coordinated. For this reason, the PlannedPTPd variant is analyzed next.

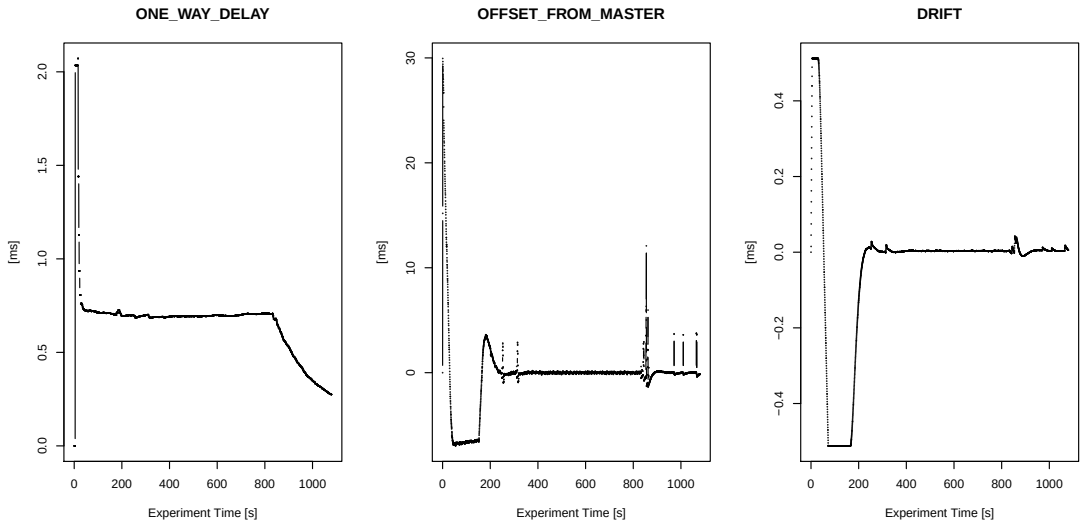


Figure 6.12.: Test Case 3; regular PTPd control system values for Subsystem 1. The initial offset from master is small. Slight disturbances may be observed at around 300 and 800 s Experiment Time.

6.4.3. PlannedPTPd with Presence of Real-Time Tasks

In the planned case, PTPd features one main thread and an additional thread for signaling the error of the cycle time to the kernel plan scheduler. Adjusting the behavior of the plan scheduler is key in order to eliminate the error inherent in the hardware clock register. The real-time tasks are now coordinated with the PTPd and the SYNC window, so less noise is expected.

Subsystem 1

The results for Test Case 3 and Subsystem 1 can be found in Figures 6.12 and 6.13. Reading from Figure 6.12, the initial offset to compensate for is significantly lesser than in the results presented in Test Cases 1 and 2. This small offset is likely due to exceptionally good synchronization achieved by running ntpdate at boot time.

It can be observed, that the levels of noise are significantly lower and that the disturbances do not affect the drift value significantly.

Figure 6.13 shows the transient plan control values corresponding to the values in Figure 6.12. The sync shift variable can be seen to start out with a significant impulse upon the activation of the plan in the Linux kernel which the plan control loop manages to dampen quickly. It can be seen that the PMP response in the beginning perfectly matches the sync shift value. It starts to drift away with changing drift input from the regular control loop to the Network Time Protocol (NTP) FLL. This is because the PMP response is measured in disciplined kernel time which is affected by the FLL.

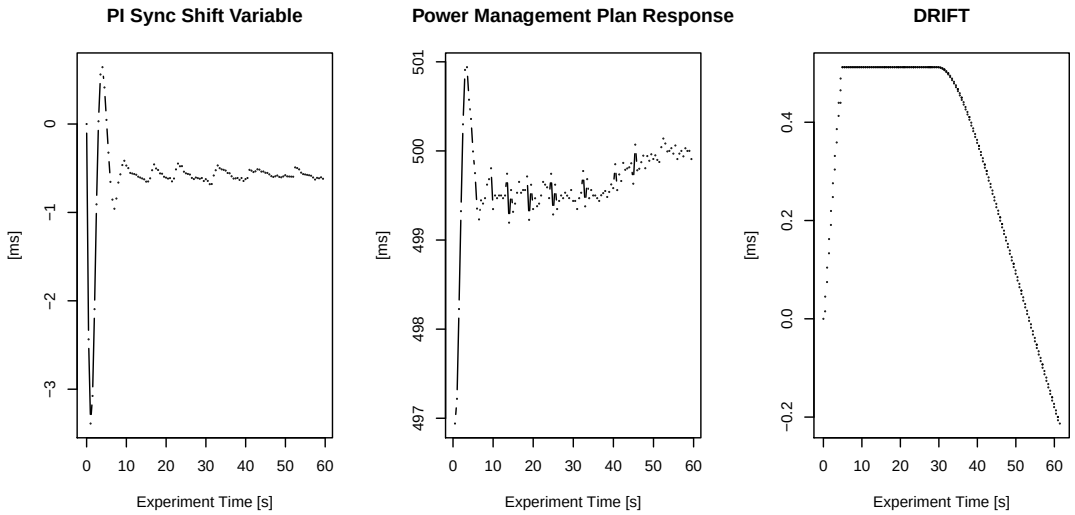


Figure 6.13.: Test Case 3; transient response of the plan control system for Subsystem 1. In the beginning, the plan scheduler response is aligned with the plan behavior, after second 40, the reponse starts to shift to 500 ms due to to the drift value for compensation of the offsets.

Subsystem 2

Figure 6.14 depicts the regular control system values for Subsystem 2 in this test case. It is clearly visible that although the offset is large in the beginning, all estimated values are subject to few noise. At first, the drift value is instructed to compensate the offset and then converges to a small value around zero indicating few systematic drift.

This is achieved by correcting the plan cycle time using the PMP node to signal the sync shift variable. In Figure 6.15 it can be seen that the sync shift starts out with a similar impulse at the beginning. Due to the negative drift value, the actual PMP response measured in disciplined kernel time is much higher.

6.5. Summary

Within this chapter, a set of experiments was conducted which are designed to:

- Test and benchmark the timing characteristics, especially under presence of dynamic Quality of Service (QoS) constraints.
- Evaluate the possible power savings while abiding QoS-constraints in lightly loaded systems.
- Test the correctness and performance of the implementation of the sequential logic operators which may be used as part of a PMP.

As a proof of concept, all these functions were benchmarked in an integration test by extending the Precision Time Protocol daemon (PTPd) and assessing the quality of

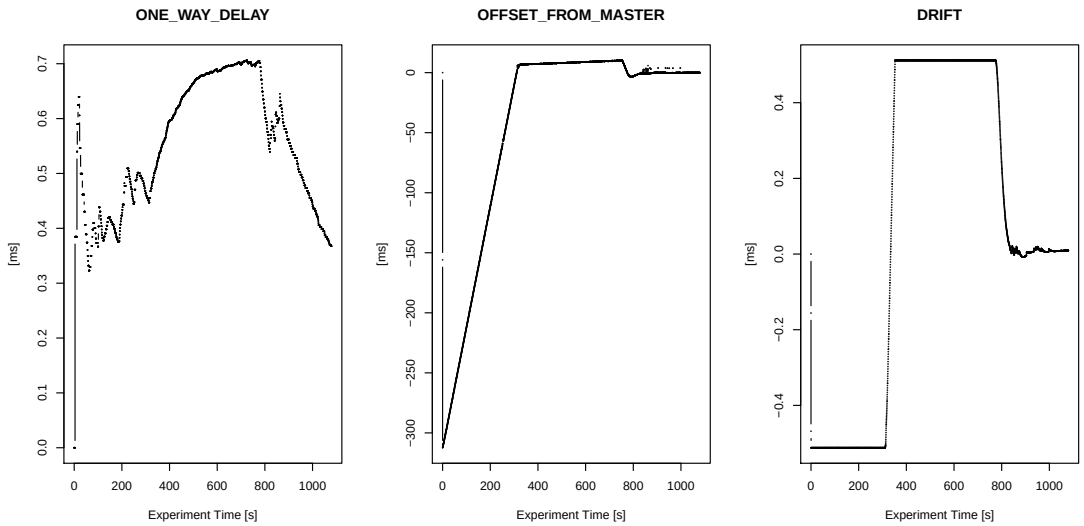


Figure 6.14.: Test Case 3; regular PTPd control system values for Subsystem 2. Initial Offset is comparably large.

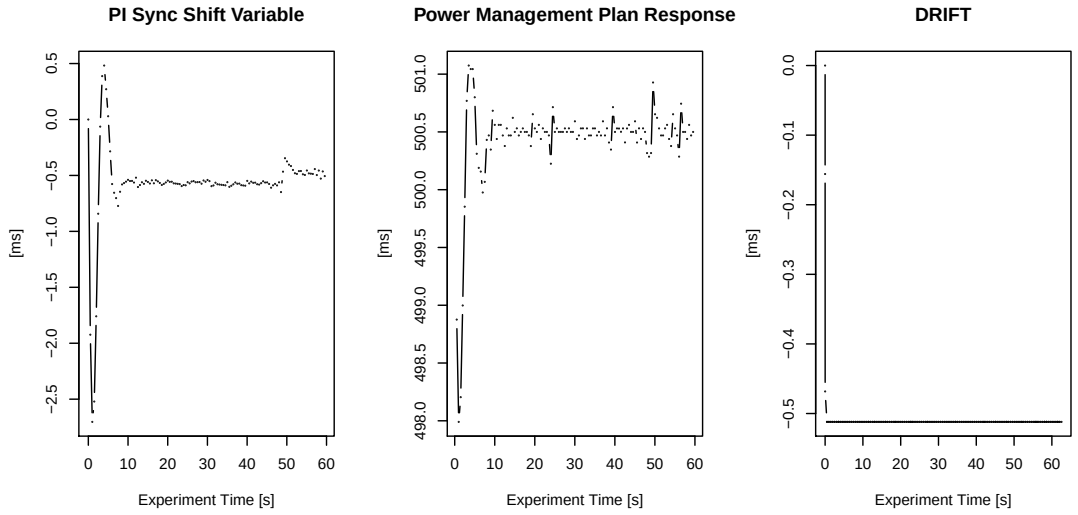


Figure 6.15.: Test Case 3; transient response of the plan control system for Subsystem 2. Sync shift, drift and response pretty much constant during the phase of systematic compensation of the offsets.

software-only synchronization achievable in a prototypical distributed system.

The extensions towards planning the execution of the PTPd demonstrate the interplay of the plan scheduler with the different kernel components and tasks on the application layer.

7. Conclusion

Current embedded systems supporting hard real-time capabilities lack a scheduling mechanism for power states. Having applications control power saving capabilities may be infeasible in the case of highly integrated systems, as they are typical for, e.g. the automotive industry. From a system integration stand point, the repartitioning of software to hardware units is a typical task. Having the operating system cope with the scheduling of software and, at least indirectly, power is desirable.

For different applications, different levels of service quality are necessary. These qualities refer to processing latency of input data and thus may affect throughput and safety critical control systems. Linux features an advanced framework for Quality of Service. It works like a middleware, which processes quality requests from applications and in the end mediates resource usage. This thesis builds upon this framework and adjusts it in a way that Quality of Service, or power states may be explicitly scheduled beforehand as part of a Power Management Plan (PMP). This allows for increasing predictability in the dynamic range of service quality and power consumption over time.

Power Management Planning

This thesis presents the concept, implementation and evaluation of a computational model from system inputs to running PMPs. These PMPs effectively allow to correlate system tasks with system power states, thus making the scheduling of power states explicitly available in the kernel scheduler. Additionally to scheduling these power states, a Boolean constraint satisfaction calculus is presented to target stability in supply voltage levels of automotive systems.

Evaluation and Synchronization

PMPs are optimized for execution directly as an operating system scheduling paradigm. As a proof of concept, a PMP scheduling discipline was integrated into the Linux 3.0 kernel. The discipline was tested on a single subsystem setup as well as on a distributed system including time synchronization. It was shown that the paradigm may enhance the Linux scheduling to be compatible with the IEEE1588 Precision Time Protocol synchronization mechanism. This mechanism as well as the traditional Network Time Protocol work on a disciplined calculative kernel time presented to the application layer.

Combining these modules, the platform may support synchronized scheduling along a distributed system, provide high levels of Quality of Service and improve power consumption values at the same time. The platform enables a multitude of open research directions.

Future Work

Future work is to effectively use and test the platform features to a real world voltage stability and energy efficiency benchmark. The flexible notion of PMP arises the question of which patterns and best practices apply to which type of application. The logging and evaluation subsystem has proven stable and powerful. It may be used for future testing and evaluation purposes.

Additionally, research on further improving system latencies or off loading the platform capabilities to dedicated hardware units can be researched. On the side of operating system implementations, concepts to combine the presented approach with virtualization and other shared computing resources need to be researched.

It is possible to apply PMPs also to computing systems of larger scale, since the presented mechanisms are integrated into the Linux kernel. They allow to work on actor-oriented embedded system tasks as well as traditional and more complex computing tasks. Evaluating them using other use cases and applications on different pieces of hardware opens up a multitude of future research directions.

Appendix

List of Figures

List of Tables

List of Abbreviations

ACPI	Advanced Configuration and Power Interface. 4
APM	Advanced Power Management. 4
AUTOSAR	AUTomotive Open System ARchitecture. 12, 97
CAN	Controller Area Network. 66, 67, 75, 77
CASE	Computer-Aided System Engineering. 9
CFQ	Completely Fair Queuing. 28
COM	Computer-On-Module. 77
DMA	Direct Memory Access. 22
ECU	Electronic Control Unit. 12, 75, 77
EDF	Earliest Deadline First. 25, 26
FIFO	First-In First-Out. 28
FLL	Frequency Locked Loop. 16, 68, 87, 89
HAL	Hardware Abstraction Layer. 19
HiL	Hardware-in-the-Loop. 10
MiL	Model-in-the-Loop. 10
NFS	Network File System. 75
NTP	Network Time Protocol. 89
PLL	Phase Locked Loop. 16
PMP	Power Management Plan. 17, 37–39, 45, 58, 61–64, 66, 67, 72, 74, 75, 80, 81, 83, 84, 87, 89, 90, 93, 94
PTP	Precision Time Protocol. 17, 29, 30, 32, 69, 72–74
PTPd	Precision Time Protocol daemon. 29, 30, 61, 68–70, 72–74, 90, 92
QoS	Quality of Service. 7, 11, 12, 17, 22–24, 63, 66, 90
RCPSP	Resource–Constraint Project–Scheduling Problem. 16
RPC	Remote Procedure Call. 6
RR	Round-Robin. 28

SysML	Systems Modeling Language. 9
TCP	Transmission Control Protocol. 6, 23
TUM	Technische Universität München. v, 75
UML	Unified Modeling Language. 9, 41

Advised Theses, Technical Reports

[Brachert 2013] Brachert, Tobias: *3D Visualization of Technical Automotive Architectures and Its Process of Running Vehicular Functions.* Technical Report: Interdisciplinary Project, Technische Universität München, 2013

[Duvnjak et al. 2013] Duvnjak, Filip ; Molotnikov, Zaur ; Nosovic, Stefan ; Stoimenov, Aleksandar: *EndorA Modeling and Simulation Framework.* Technical Report: Interdisciplinary Project, Technische Universität München, 2013

[Enger 2013] Enger, Andre: *Evaluation of Function Partitioning for Distributed Automotive Systems.* Technical Report: Interdisciplinary Project, Technische Universität München, 2013

[Engeser 2013] Engeser, Benedikt: *Deployment, Scheduling, and I/O of Physical Simulation Models in Distributed Systems.* Bachelor's thesis, Technische Universität München, 2013

[Fuchs 2012] Fuchs, Andreas: *Static Multicasting Respecting Quality of Service over CAN and Ethernet Targeting a LabVIEW-based Test Bench.* Bachelor's thesis, Technische Universität München, 2012

[Fuchs 2013] Fuchs, Andreas: *Measurement and Evaluation of a Communication Middleware for Distributed Embedded Systems.* Technical Report: Interdisciplinary Project, Technische Universität München, 2013

[Gabriel 2012] Gabriel, Dirk: *Power–Management–Plan basierter Scheduler im ITE–Simulationsframework.* Bachelor's thesis, Technische Universität München, 2012

[Gleixner 2011] Gleixner, Robert: *A Linux Scheduler for Power Management in Soft Real-time Systems.* Bachelor's thesis, Technische Universität München, 2011

[Gleixner 2013] Gleixner, Robert: *Power Management Planning in Linux for Real-Time Software in a Distributed Test Bench Platform.* Technical Report: Interdisciplinary Project, Technische Universität München, 2013

[Lowinski 2013] Lowinski, Martin: *Analysis and Enhancements of Scheduling-Strategies for Multicore Systems.* Master's thesis, Technische Universität München, 2013

[Schlenk 2012] Schlenk, Alexander: *Development and Initial Startup of a Hardware Platform for Autonomous Shutdown Mechanisms in Automotive Electrical Systems.* Bachelor's thesis, Technische Universität München, 2012

[Totakura 2012] TOTAKURA, Sree H.: *Development of an Interpreter for the Distributed Execution of Software Components on an Embedded Platform.* Technical Report: Interdisciplinary Project, Technische Universität München, 2012

Own Work

[Barthels et al. 2012a] Barthels, Andreas ; Fröschl, Joachim ; Baumgarten, Uwe: An Architecture for Power Management in Automotive Systems. In: *Proceedings of the 25th International Conference on Architecture of Computing Systems (ARCS)*, 2012

[Barthels et al. 2012b] Barthels, Andreas ; Michel, Hans-Ulrich ; Walla, Gregor: Jedes Watt zählt: Intelligentes Energiemanagement für die Autos von Morgen. In: *Elektronik Automotive*. 2012, pp. 24–28

[Barthels et al. 2012c] Barthels, Andreas ; Ruf, Florian ; Schlenk, Alexander ; Walla, Gregor ; Michel, Hans-Ulrich ; Baumgarten, Uwe: PREcup-1: An Embedded System Platform for Prototyping ECU Power Management. In: *Proceedings of the 8th IEEE Vehicle Power and Propulsion Conference (VPPC)*, 2012

[Barthels et al. 2011] Barthels, Andreas ; Ruf, Florian ; Walla, Gregor ; Fröschl, Joachim ; Michel, Hans-Ulrich ; Baumgarten, Uwe: A Model for Sequence Based Power Management in Cyber Physical Systems. In: *1st International Conference on ICT as Key Technology for the Fight against Global Warming – ICT-GLOW*, 2011, pp. 87–101

[Caliskan et al. 2007] Caliskan, Murat ; Barthels, Andreas ; Scheuermann, Björn ; Mauve, Martin: Predicting Parking Lot Occupancy in Vehicular Ad Hoc Networks. In: *Proceedings of the 65th IEEE Vehicular Technology Conference*, 2007

[Jain et al. 2010] Jain, Dominik ; Barthels, Andreas ; Beetz, Michael: Adaptive Markov Logic Networks: Learning Statistical Relational Models with Dynamic Parameters. In: *Proceedings of the 19th European Conference on Artificial Intelligence*, 2010

[Lochert et al. 2005] Lochert, Christian ; Caliskan, Murat ; Scheuermann, Björn ; Barthels, Andreas ; Cervantes, Alfonso ; Mauve, Martin: Multiple Simulator Interlinking Environment for Inter Vehicle Communication. In: *Proceedings of the Second International ACM Workshop on Vehicular Ad Hoc Networks*, 2005

[Ruf et al. 2013a] Ruf, Florian ; Barthels, Andreas ; Walla, Gregor ; Winter, Michael ; Fröschl, Joachim ; Michel, Hans-Ulrich ; Herzog, Hans-Georg: Prototypical Platform and Test Bench for Investigating Automotive Energy and Power Management Paradigms. In: *Elektrik/Elektronik in Hybrid- und Elektrofahrzeugen und elektrisches Energiemanagement*, Haus der Technik, April 2013

[Ruf et al. 2012a] Ruf, Florian ; Barthels, Andreas ; Walla, Gregor ; Winter, Michael ; Kohler, Tom P. ; Michel, Hans-Ulrich ; Fröschl, Joachim ; Herzog,

Hans-Georg: Autonomous Load Shutdown Mechanism as a Voltage Stabilization Method in Automotive Power Nets. In: *Proceedings of the 8th IEEE Vehicle Power and Propulsion Conference (VPPC)*, 2012

[Ruf et al. 2012b] Ruf, Florian ; Neiss, Alexander ; Barthels, Andreas ; Kohler, Tom P. ; Michel, Hans-Ulrich ; Fröschl, Joachim ; Herzog, Hans-Georg: Design Optimization of a 14 V Automotive Power Net Using a Parallelized DIRECT Algorithm in a Physical Simulation. In: *Proceedings of the 13th International Conference on Optimization of Electrical and Electronic Equipment (OPTIM)*, 2012

[Ruf et al. 2013b] Ruf, Florian ; Schill, Markus ; Barthels, Andreas ; Kohler, Tom P. ; Michel, Hans-Ulrich ; Fröschl, Joachim ; Herzog, Hans-Georg: Topology and Design Optimization of a 14 V Automotive Power Net using a Modified Discrete PSO in a Physical Simulation. In: *Proceedings of the 9th IEEE Vehicle Power and Propulsion Conference (VPPC)*, 2013

[Walla et al. 2012a] Walla, Gregor ; Barthels, Andreas ; Ruf, Florian ; Dörfel, Robert ; Michel, Hans-Ulrich ; Fröschl, Joachim ; Sirch, Ottmar ; Baumgarten, Uwe ; Herzog, Hans-Georg ; Herkersdorf, Andreas: Framework and Model for the Evaluation of Energy Efficiency of Partitioning Alternatives. In: *Elektrik/Elektronik in Hybrid- und Elektrofahrzeugen und elektrisches Energiemanagement*, Haus der Technik, April 2012, pp. 151–158

[Walla et al. 2012b] Walla, Gregor ; Barthels, Andreas ; Ruf, Florian ; Dörfel, Robert ; Michel, Hans-Ulrich ; Fröschl, Joachim ; Sirch, Ottmar ; Baumgarten, Uwe ; Herzog, Hans-Georg ; Stechele, Walter ; Herkersdorf, Andreas: Aspects of Function Partitioning in Respect to Power Management. In: *Proceedings of the 2nd International Energy Efficient Vehicles Conference (EEVC)*, 2012

[Walla et al. 2012c] Walla, Gregor ; Gabriel, Dirk ; Barthels, Andreas ; Ruf, Florian ; Michel, Hans-Ulrich ; Herkersdorf, Andreas: ITEsim: A Simulator and Power Evaluation Framework for Electric/Electronic Architectures. In: *Proceedings of the 8th IEEE Vehicle Power and Propulsion Conference (VPPC)*, 2012

[Walla et al. 2013] Walla, Gregor ; Molotnikov, Zaur ; Barthels, Andreas ; Michel, Hans-Ulrich ; Stechele, Walter ; Herkersdorf, Andreas: A Design Space Exploration Framework or Automotive Embedded Systems and Their Power Management. In: *Proceedings of the 27th European Conference on Modeling and Simulation (ECMS 2013)*, 2013

Bibliography

[ACPI 2013] *Advanced Configuration and Power Interface.* http://acpi.info, 2013

[Alur and Dill 1994] Alur, Rajeev ; Dill, David L.: A theory of timed automata. In: *Theoretical Computer Science* 126 (1994), Nr. 2, 183 - 235. http://dx.doi.org/http://dx.doi.org/10.1016/0304-3975(94)90010-8. – DOI http://dx.doi.org/10.1016/0304–3975(94)90010–8. – ISSN 0304–3975

[ANETTEST 2012] *AnetTest - automated testing of network devices and applications.* 2012. – http://anettest.sourceforge.net/

[ARM Ltd. 2010] ARM Ltd.: *ARM Cortex Applications Processor A8: Technical Reference Manual.* http://www.arm.com/products/processors/cortex-a/cortex-a8.php?tab=Resources+, 2010

[Aurrecoechea et al. 1998] Aurrecoechea, Cristina ; Campbell, Andrew T. ; Hauw, Linda: A survey of QoS architectures. In: *Multimedia systems* 6 (1998), Nr. 3, pp. 138–151

[AUTOSAR 2013] *Automotive Open System Architecture.* http://www.autosar.org/, 2013

[Balaguer et al. 2012] Balaguer, Sandie ; Chatain, Thomas ; Haar, Stefan: A concurrency-preserving translation from time Petri nets to networks of timed automata. In: *Formal Methods in System Design* 40 (2012), Nr. 3, 330-355. http://dx.doi.org/10.1007/s10703-012-0146-4. – DOI 10.1007/s10703–012–0146–4. – ISSN 0925–9856

[Behrmann et al. 2006] Behrmann, G. ; David, A. ; Larsen, K.G. ; Hakansson, J. ; Petterson, P. ; Yi, Wang ; Hendriks, M.: UPPAAL 4.0. In: *Quantitative Evaluation of Systems, 2006. QEST 2006. Third International Conference on*, 2006, pp. 125–126

[Benini et al. 2000] Benini, L. ; Bogliolo, A. ; De Micheli, Giovanni: A Survey of Design Techniques for System-Level Dynamic Power Management. In: *IEEE Transactions on very large scale integration (VLSI) systems* 8 (2000), June, Nr. 3

[BMW Technology Guide 2014] BMW Group: *BMW Technology Guide: Wiring Harness.* http://www.bmw.com/com/en/insights/technology/technology_guide/articles/wiring_harness.html. Version: February 2014

[Bucher et al. 2013] Bucher, Roberto ; Dozio, Lorenzo ; Gasperini, Daniele ; Mayer, Hannes ; Mantegazza, Paolo ; Masarati, Pierangelo ; Neuhauser, Michael ; Racciu, Giovanni ; Schleef, David ; Soetens, Peter: *RTAI - the RealTime Application Interface for Linux from DIAPM.* 2013. – http://www.rtai.org/

[Calandrino et al. 2006] Calandrino, John M. ; Leontyev, Hennadiy ; Block, Aaron ; Devi, UmaMaheswari C. ; Anderson, James H.: LITMUS^{RT}: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. In: *Real-Time Systems Symposium, 2006. RTSS'06. 27th IEEE International* IEEE, 2006, pp. 111–126

[Ceraolo 2000] Ceraolo, M.: New dynamical models of lead-acid batteries. In: *Power Systems, IEEE Transactions on* 15 (2000), November, Nr. 4, pp. 1184–1190

[Cerqueira and Brandenburg 2013] Cerqueira, Felipe ; Brandenburg, Björn B: A Comparison of Scheduling Latency in Linux, PREEMPT RT, and LITMUS^{RT}. In: *OSPERT 2013* (2013), pp. 20

[Cochran and Marinescu 2010] Cochran, Richard ; Marinescu, Cristian: Design and implementation of a PTP clock infrastructure for the Linux kernel. In: *Precision Clock Synchronization for Measurement Control and Communication (ISPCS), 2010 International IEEE Symposium on* IEEE, 2010, pp. 116–121

[Cochran et al. 2011] Cochran, Richard ; Marinescu, Cristian ; Riesch, Christian: Synchronizing the Linux system time to a PTP hardware clock. In: *Precision Clock Synchronization for Measurement Control and Communication (ISPCS), 2011 International IEEE Symposium on* IEEE, 2011, pp. 87–92

[Colak et al. 2013] Colak, Selcuk ; Agarwal, Anurag ; Erenguc, Selcuk: Multi-Mode Resource-Constrained Project-Scheduling Problem With Renewable Resources: New Solution Approaches. In: *Journal of Business & Economics Research (JBER)* 11 (2013), Nr. 11, pp. 455–468

[Correll et al. 2005] Correll, Kendall ; Barendt, Nick ; Branicky, Michael: Design considerations for software only implementations of the IEEE 1588 precision time protocol. In: *Conference on IEEE* Bd. 1588, 2005, pp. 11–15

[Dick et al. 1998] Dick, Robert P. ; Rhodes, David L. ; Wolf, Wayne: TGFF: task graphs for free. In: *Proceedings of the 6th international workshop on Hardware/software codesign* IEEE Computer Society, 1998, pp. 97–101

[Eugster et al. 2003] Eugster, Patrick T. ; Felber, Pascal A. ; Guerraoui, Rachid ; Kermarrec, Anne-Marie: The many faces of publish/subscribe. In: *ACM Computing Surveys (CSUR)* 35 (2003), Nr. 2, pp. 114–131

[Faggioli et al. 2009] Faggioli, Dario ; Checconi, Fabio ; Trimarchi, Michael ; Scordino, Claudio: An EDF scheduling class for the Linux kernel. In: *Proc. of the Real-Time Linux Workshop*, 2009

[Fedorova et al. 2007] Fedorova, Alexandra ; Seltzer, Margo ; Smith, Michael D.: Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler. In: *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*. Washington, DC, USA : IEEE Computer Society, 2007 (PACT '07). – ISBN 0–7695–2944–5, pp. 25–38

[Fuchs et al. 2010] Fuchs, Martin ; Scheer, Patrick ; Grzemba, Andreas: Selektiver Teilnetzbetrieb im Fahrzeug: Eine Realisierung für den CAN-Bus und Adaption auf andere Bussysteme. In: *AmE 2010 - Automotive meets Electronics (GMM-FB 64)*, 2010

[Gehring et al. 2009] Gehring, R. ; Fröschl, J. ; Kohler, T.P. ; Herzog, H.-G.: Modeling of the automotive 14 V power net for voltage stability analysis. In: *Vehicle Power and Propulsion Conference*, 2009, pp. 71–77

[Gehring and Herzog 2009] Gehring, R. ; Herzog, H.-G.: Simulation der Spannungsstabilität im 12 V Energiebordnetz bei komplexen E/E-Architekturen. In: *Moderne Elektronik im Kraftfahrzeug, Tagung Elektronik im Kraftfahrzeug.* Dresden : Haus der Technik e.V., June 2009

[Godard 2012] Godard, Sebastien: *SYSSTAT Utilities Version 9.0.6.* 2012. – http://sebastien.godard.pagesperso-orange.fr/

[GUMSTIX, Inc. 2012] GUMSTIX, Inc.: *Computers-on-Module: Overo.* http://www.gumstix.com/, 2012

[Haberl 2011] Haberl, Wolfgang: *Code Generation and System Integration of Distributed Automotive Applications*, Technische Universität München, Diss., 2011

[Hartkopp and Thürmann 2005] Hartkopp, Oliver ; Thürmann, Urs: *Interface for communications between vehicle applications and vehicle bus systems, implements at least one vehicle bus protocol within protocol family, between socket-and network layers.* November 17 2005. – DE Patent 102,004,020,880

[Hartkopp et al. 2013] Hartkopp, Oliver ; Thürmann, Urs ; Venzano, Daniele: *Low-level CAN interface–Application Programmers Interface.* 2013. – http://www.brownhat.org/docs/socketcan/llcf-api.html

[Irani et al. 2003] Irani, S. ; Shukla, S. ; Gupta, R.: Online Strategies for Dynamic Power Management in Systems with Multiple Power-Saving States. In: *ACM Transactions on Embedded Computing Systems* 2 (2003), August, Nr. 3, pp. 325–346

[Jackson et al. 2013] Jackson, Daniel ; Milicevic, Aleksandar ; Torlak, Emina ; Kang, Eunsuk ; Near, Joe: *alloy: a language & tool for relational models.* http://alloy.mit.edu, 2013

[Jejurikar and Gupta 2004] Jejurikar, R. ; Gupta, R.: Dynamic Voltage Scaling for Systemwide Energy Minimization in Real-Time Embedded Systems. In: *ISLPED*, 2004

[Karsai et al. 2003] Karsai, G. ; Sztipanovits, J. ; Ledeczi, A. ; Bapty, T.: Model-integrated development of embedded software. In: *Proceedings of the IEEE* 91 (2003), Januar, Nr. 1, pp. 145 – 164. http://dx.doi.org/10.1109/JPROC.2002.805824. – DOI 10.1109/JPROC.2002.805824. – ISSN 0018–9219

[Katoen et al. 2013] Katoen, Joost-Pieter ; Noll, Thomas ; Wu, Hao ; Santen, Thomas ; Seifert, Dirk: Model-based energy optimization of automotive control systems. In: *Proceedings of the Conference on Design, Automation and Test in Europe* EDA Consortium, 2013, pp. 761–766

[Kim et al. 2002] Kim, Hyun-Jun ; Park, Sang-Hyun ; Kim, Jung-Guk ; Kim, Moon-Hae ; Rim, Kee-Wook: TMO-Linux: a Linux-based real-time operating system supporting execution of TMOs. In: *Object-Oriented Real-Time Distributed Computing, 2002. (ISORC 2002). Proceedings. Fifth IEEE International Symposium on*, 2002, pp. 288–294

[Kim et al. 2005] Kim, Jung-Guk ; Kim, Moon-Hae ; Kim, Kwang ; Heu, Shin: TMO-eCos: an eCos-based real-time micro operating system supporting execution of a TMO structured program. In: *Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on*, 2005, pp. 182–189

[Kohler 2013] Kohler, Tom P.: *Prädiktives Leistungsmanagement in Fahrzeugbordnetzen*, Technische Universität München, Diss., 2013

[Kohler et al. 2010] Kohler, Tom P. ; Fröschl, Joachim ; Bertram, Christiane ; Buecherl, Dominik ; Herzog, Hans-Georg: Approach of a Predictive, Cybernetic Power Distribution Management. In: *The 25th World Electric Vehicle Symposium and Exposition*, World Electric Vehicle Association (WEVA), November 2010

[Kohler et al. 2011] Kohler, Tom P. ; Gehring, Rainer ; Fröschl, Joachim ; Buecherl, Dominik ; Herzog, Hans-Georg: Voltage Stability Analysis of Automotive Power Nets based on Modeling and Experimental Results. In: Chiaberge, Marcello (Hrsg.): *New Trends and Developments in Automotive System Engineering*. InTech, 2011, Chapter 30, pp. 611–630. – Available from: http://www.intechopen.com/articles/show/title/voltage-stability-analysis-of-automotive-power-nets-based-on-modeling-and-experimental-results

[Kovácsházy and Ferencz 2012] Kovácsházy, Tamás ; Ferencz, Bálint: Performance evaluation of PTPd, a IEEE 1588 implementation, on the x86 Linux platform for typical application scenarios. In: *Instrumentation and Measurement Technology Conference (I2MTC), 2012 IEEE International* IEEE, 2012, pp. 2548–2552

[Levis et al. 2005] Levis, P. ; Madden, S. ; Polastre, J. ; Szewczyk, R. ; Whitehouse, K. ; Woo, A. ; Gay, D. ; Hill, J. ; Welsh, M. ; Brewer, E. ; Culler, D.: TinyOS: An Operating System for Sensor Networks. Version: 2005. http://dx.doi.org/10.1007/3-540-27139-2_7. In: Weber, Werner (Hrsg.) ; Rabaey, JanM. (Hrsg.) ; Aarts, Emile (Hrsg.): *Ambient Intelligence*. Springer Berlin Heidelberg, 2005. – DOI 10.1007/3–540–27139–2_7. – ISBN 978–3–540–23867–6, 115-148

[Linaro 2012] *Linaro TI Kernel*. 2012. – http://www.linaro.org/

[Mazumder et al. 2012] Mazumder, Biswajit ; Jiang, Hao ; Hallstrom, Jason O.: Design and Analysis of Almost-Always-Sleeping Schedulers for Embedded Systems. In: *The Sixth International Conference on Sensor Technologies and Applications*

(SENSORCOMM), 2012, pp. 284–291. – http://www.thinkmind.org/download.php?articleid=sensorcomm_2012_12_20_10202

[Menage 2006] Menage, Paul: *CGROUPS*. 2006. – https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt

[Microsoft Corp. 1996] Microsoft Corp.: *Advanced Power Management Specification*. http://download.microsoft.com/download/1/6/1/161ba512-40e2-4cc9-843a-923143f3456c/APMV12.rtf, 1996

[Microsoft FORMULA 2012] *FORMULA – Modeling Foundations*. http://research.microsoft.com/en-us/projects/formula/, 2012

[Mills et al. 2010] Mills, D. ; Delaware, U. ; Martin, J.: *Network Time Protocol Version 4: Protocol and Algorithms Specification*. http://tools.ietf.org/html/rfc5905. Version: 2010

[Molnar 2013] Molnar, Ingo: *Linux RT-Preempt Patch*. 2013. – http://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO

[Moore 1956] Moore, E.F.: Gedanken-experiments on sequential machines. In: *Automata studies* 34 (1956), pp. 129–153

[Moreno and de Niz 2012] Moreno, G.A. ; Niz, D. de: An Optimal Real-Time Voltage and Frequency Scaling for Uniform Multiprocessors. In: *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2012 IEEE 18th International Conference on*, 2012. – ISSN 1533–2306, pp. 21–30

[Nelson 2012] Nelson, Phil: *bc - an arbitrary precision calculator language*. 2012. – http://www.gnu.org/software/bc/

[Object Management Group 2011] Object Management Group: State Machines. In: *Unified Modeling Language Super Structure*. 2011, Chapter 15, pp. 535–592. – http://www.omg.org/spec/UML/2.4.1/

[Ohly et al. 2008] Ohly, Patrick ; Lombard, David N. ; Stanton, Kevin B.: Hardware assisted precision time protocol. Design and case study. In: *Proceedings of LCI International Conference on High-Performance Clustered Computing. Urbana, IL, USA: Linux Cluster Institute* Bd. 5, 2008, pp. 121–131

[Oikawa and Rajkumar 1998] Oikawa, Shui ; Rajkumar, Raj: Linux/RK: A portable resource kernel in Linux. In: *IEEE Real-Time Systems Symposium* Citeseer, 1998

[Polenov et al. 2007] Polenov, D. ; Probstle, H. ; Brosse, A. ; Domorazek, G. ; Lutz, J.: Integration of supercapacitors as transient energy buffer in automotive power nets. In: *Power Electronics and Applications, European Conference on*, 2007, pp. 1–10

[Ramchandani 1974] Ramchandani, Chander: *Analysis of asynchronous concurrent systems by timed Petri nets*, Massachusetts Institute of Technology, Diss., 1974

[Reinhardt and Kucera 2013] Reinhardt, Dominik ; Kucera, Markus: Domain Controlled Architecture - A New Approach for Large Scale Software Integrated Automotive Systems. In: *PECCS*, 2013, pp. 221–226

[Rittmann 2008] Rittmann, Sabine: *A methodology for modeling usage behavior of multi-functional systems*, Technische Universität München, Diss., 2008

[Schantz et al. 2003] Schantz, Richard E. ; Loyall, Joseph P. ; Rodrigues, Craig ; Schmidt, Douglas C. ; Krishnamurthy, Yamuna ; Pyarali, Irfan: Flexible and Adaptive QoS Control for Distributed Real-Time and Embedded Middleware. Version: 2003. http://dx.doi.org/10.1007/3-540-44892-6_19. In: Endler, Markus (Hrsg.) ; Schmidt, Douglas (Hrsg.): *Middleware 2003* Bd. 2672. Springer Berlin Heidelberg, 2003. – DOI 10.1007/3–540–44892–6_19. – ISBN 978–3–540–40317–3, 374-393

[Schmutzler et al. 2010] Schmutzler, Christoph ; Kruger, Andreas ; Schuster, Fred ; Simons, Martin: Energy Efficiency in Automotive Networks: Assessment and Concepts. In: *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, 2010, pp. 232–240

[Silberschatz et al. 2005] Silberschatz, Abraham ; Gagne, Greg ; Galvin, Peter B.: *Operating System Concepts*. Seventh. John Wiley and Sons, Inc., 2005

[Stuijk et al. 2006] Stuijk, Sander ; Geilen, Marc ; Basten, Twan: SDF^ 3: SDF For Free. In: *Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on* IEEE, 2006, pp. 276–278

[TCPDump 2012] *TCPDump Version 4.1.1*. 2012. – http://www.tcpdump.org/

[Texas Instruments 2012] Texas Instruments: *OMAP Power Management Whitepaper*. 2012. – http://www.ti.com/lit/an/sprt495/sprt495.pdf

[Tiwari 2008] Tiwari, Ashish: Abstractions for hybrid systems. In: *Formal Methods in System Design* 32 (2008), Nr. 1, 57-83. http://dx.doi.org/10.1007/s10703-007-0044-3. – DOI 10.1007/s10703–007–0044–3. – ISSN 0925–9856

[Vector Informatik GmbH 2013] Vector Informatik GmbH: *PREEvision*. http://vector.com/vi_preevision_en.html, July 2013

[Šimunić et al. 1999] Šimunić, Tajana ; Benini, Luca ; De Micheli, Giovanni: Cycle-accurate simulation of energy consumption in embedded systems. In: *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*. New York, NY, USA : ACM, 1999 (DAC '99). – ISBN 1–58113–109–7, 867–872

[Weibel 2009] Weibel, Hans: Technology Update on IEEE 1588: The Second Edition of the High Precision Clock Synchronization Protocol. In: *Embedded World* (2009)

[Xenomai 2013] *Xenomai: Real-Time Framework for Linux*. 2013. – http://www.xenomai.org/

[Zhu 2013] Zhu, Wensi: IEEE 1588 implementation with FLL vs. PLL. In: *Precision Clock Synchronization for Measurement Control and Communication (ISPCS), 2013 International IEEE Symposium on* IEEE, 2013, pp. 71–76