embeddedworld 2014
Exhibition&Conference
... it´s a smarter world

Operating System Concepts for Embedded Multicores

Oliver Horst, Adriaan Schmidt
Fraunhofer Institute for Embedded Systems and Communication Technologies ESK
Munich, Germany
{oliver.horst, adriaan.schmidt}@esk.fraunhofer.de

***Abstract*—Currently we can see an increasing adoption of multi-core platforms in the area of embedded systems. While these new hardware platforms offer the potential to satisfy the ever increasing demand for computational power, they pose considerable challenges with regard to software development. This affects the application software itself, but also the system design and architecture. Here, we address the consequences for operating system architecture in embedded systems. After discussing current approaches, we present our own proposal for a flexible and configurable operating system design, targeted at embedded multi-core platforms.**

Keywords—operating systems; multi-core; AMP; SMP

I. INTRODUCTION

At the moment, we can see a trend of increasing complexity in embedded systems. In some instances the tasks they perform become more computation-intensive, like image processing frequently used in advanced driver-assistance systems; other systems face increasing data volume and throughput requirements, e.g. current and next generation wired and wireless communication systems. Often we also find that embedded systems can no longer be seen as isolated. Instead we have to consider their interactions with other devices they are linked to, or regard them as a "system of systems". All of these factors introduce new requirements when designing embedded systems. Aspects like dynamic re-configuration of the system at runtime, or the integration of different applications of mixed criticality on one device are only few examples of new challenges. One of the consequences of these trends is that future embedded systems will no longer be dedicated to only one static task for which they are programmed and optimized at design time. Instead they will need to be flexible and open, able to adapt to changing scenarios, while still meeting the timing requirements of their respective applications.

In this context, multi-core platforms play an important role. They can satisfy the demand for computational power, and can provide the flexibility needed in future embedded systems. However, multi-core technology itself introduces new difficulties and challenges with regard to application and system software.

The biggest of these challenges is that the software must explicitly target parallel execution in order to benefit from the available processing power. This means that the software must be parallelized and then distributed among the cores. While this applies to all parallel and multi-core platforms, embedded systems introduce even more constraints. Because factors like fabrication cost and energy consumption come into play, embedded multi-core devices are often specialized Systems-on-Chip (SoCs) with heterogeneous hardware architectures. In addition, embedded software must often meet certain timing-related constraints. Because of the high complexity of the systems, it is hard to guarantee compliance with these real-time requirements.

The use of operating systems can help alleviate these problems by reducing the complexity that is visible to the software developer. By using abstract programming interfaces, the developer does not need concern himself with the details of the hardware platform. But as embedded platforms and applications become more complex, there is need for more specialized operating systems.

In this paper we present a new operating system design concept that has two main features: First, we use a system layout that simultaneously permits both static task placement for deterministic timing and a high degree of flexibility at runtime to optimally utilize the available resources. Second, we provide a modular and configurable operating system, in which main functions can be tailored to the specific application.

In Section II we outline current approaches to embedded multi-core operating systems. We present our idea of a flexible operating systems design in Section III. In Section IV we present two different embedded application scenarios to which we have tailored our configurable architecture. Section V concludes the paper and gives an outlook on future research.

The research leading to these results has received funding from the German Federal Ministry for Economic Affairs and Energy and the Bavarian Ministry of Economic Affairs and Media, Energy and Technology.

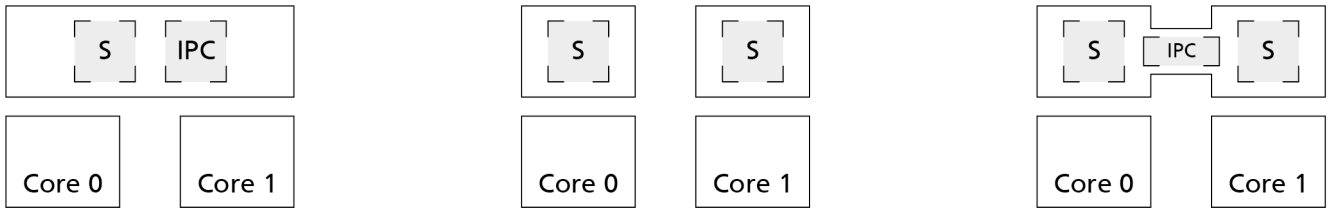

Fig. 1. Illustration of different software deployment strategies for multi-core platforms, and the location of the key components scheduling (S) and inter-process communication (IPC): (a) Symmetric Multiprocessing (SMP), (b) Asymmetric Multiprocessing (AMP), and (c) our combined approach.

II. SOFTWARE DEPLOYMENT STRATEGIES FOR MULTI-CORE PLATFORMS

In general there are two major approaches with regard to the operating system architecture for multi-core platforms: symmetric multiprocessing (SMP) and asymmetric multiprocessing (AMP). In the following we use the terms *SMP* and *AMP* to refer to software concepts, independent of the underlying hardware platform. This should not be confused with the terms *homogeneous* and *heterogeneous*, which we use to describe whether a multi-core platform has identical cores (homogeneous) or cores of different architectures or with different features (heterogeneous).

In SMP systems there is one global instance of the operating system that controls and manages all cores. Scheduling is performed globally and centrally, as well as load balancing and control of I/O devices. Having only one operating system instance on a multi-core system means that operating system code and data structures are shared among multiple cores. As a consequence SMP operating systems are only suitable for homogeneous platforms with a shared memory hardware architecture. From the application programmer's point of view, SMP systems are easy to use. To leverage the parallel hardware platform, only a relatively simple API is used (e.g. pthreads), while the distribution of tasks to cores and the management of resources is handled dynamically by the operating system. However, this flexibility and scalability has drawbacks when considering embedded systems. One problem is non-determinism in the timing of task execution, which is a side-effect of the complexity of dynamic scheduling on multiple cores. Also the usage of shared data structures within the kernel, which need to be protected by synchronization mechanism like locks or semaphores, can lead to unforeseen delays in program execution.

An alternative to SMP is AMP, where the cores operate independently. They can either execute separate instances of the same operating system, or run totally different environments (e.g. a mix of different operating systems, including bare-metal applications). The different operating system instances do not share any data structures, so this approach is also suited for heterogeneous platforms or devices without shared memory. Software development and system integration of AMP systems differs from that of SMP platforms. In an AMP system, many decisions on task distribution and resource allocation are taken at design-time, and cannot change dynamically while the system is running. While this requires additional planning effort and can make application development and deployment less flexible, it also has advantages. The inherently higher determinism permits easier analysis and verification of the system's behavior, particularly with regard to timing.

Pure SMP architectures have limitations when it comes to their deployment in embedded systems. They require homogeneous, shared-memory hardware platforms, and they introduce sources of non-determinism. On the other hand, AMP architectures lack the required runtime flexibility to construct next-generation embedded designs that are open and address aspects like re-configuration or mixed-criticality applications. Thus, our aim is to design an operating system architecture that leverages the advantages of both approaches while addressing their disadvantages. Similar approaches combining SMP and AMP concepts can also be found in operating systems like Barrelfish [*1*] or Helios [*2*]. However, our proposal specifically targets embedded systems with their constraints with regard to hardware platforms and application requirements.

III. MUC-OS: A FLEXIBLE OPERATING SYSTEM DESIGN

We propose an operating system concept that bridges the gap between SMP and AMP. Our design is based on a single-core operating system which we deploy in an AMP configuration. Each core executes its own instance of the kernel; there are no shared data structures within the kernel. This significantly reduces run-time complexity, and thus facilitates analysis of timing behavior and resource usage. To enable cooperation of the independent OS instances we provide a message-based mechanism for inter-process communication (IPC). Our proposed design and its relation to SMP and AMP concepts is illustrated in Fig. 1. Thanks to the message-based communication mechanism it is possible to provide a distributed execution environment similar to SMP systems, where services and applications can be executed distributed across multiple processor cores. In fact, from the application's point of view, our proposed system design is indistinguishable from an SMP system.

We implemented the proposed operating system design in form of a configurable operating system prototype for embedded systems: MUC-OS. Our new operating system is configurable, because we believe that the requirements of embedded applications are so diverse that they cannot be met by a single configuration or even implementation of an operating system. Thus, we propose a modular kernel design that defines only small parts of the overall architecture and concept. The goal of the design is to establish MUC-OS as a flexible operating system platform and basis for further research.

TABLE I. OVERVIEW ON THE MODULE STRUCTURE OF MUC-OS.

	Module name	Description
core functionality	Boot Code	The Boot Code systematically initializes the hardware with help of the Hardware Abstraction Layer. It brings the hardware platform to a defined state in which all processor cores are correctly initialized, booted with their own instance of the operating system kernel and connected via the IPC module. The implementation of this module is fixed.
	Process Management	This module defines the task state model as shown in Fig. 2 and provides functions to create and destroy tasks, and to migrate them between processor cores. Our state model is based on the five-state model originally presented in [*5*], with an extension that allows for a task to suspend other tasks. The implementation of this module is fixed.
	Module Interfaces	The interface definition is not strictly a module, but still forms a component of the overall MUC-OS kernel. It specifies the interfaces of all modules listened in this table, as well as their interconnections.
essential modules	HardwareAabstraction Layer (HAL)	The HAL provides functions to save/restore a task context, enable/disable interrupts, initialize task stacks, and to initialize specific hardware components. Additionally, the HAL implements the system call interfaces and manages the context switch to the kernel in case of a system call. The user has to provide an own HAL module for each hardware platform he wants to support.
	Scheduler	The scheduler handles the local management of computational resources. Each processor core has its own scheduler; a coordination between the instances can take place through the exchange of IPC messages. Thus, also a global scheduling using task migrations between cores can be realized. Thanks to the modular structure, even a hierarchical scheduler is possible that itself utilizes other scheduler modules to schedule the individual containers.
	Inter-Process Communication (IPC)	The IPC module implements the message passing paradigm within the MUC-OS kernel. It provides a message based communication mechanism for inter-process and inter-core communication. Implementations of the IPC module could either be a simple queue based message exchange protocol, more sophisticated protocols like e.g. MCAPI [*16*], or any other mechanism that could be used to deliver messages.
optional modules	Memory Manager	If required by an application, the memory manager module implements memory allocation and deallocation functions. A possible implementation is e.g. a memory pool based management. The module may also be extended or adapted by other modules to provide features like garbage collection.
	Resource Manager	This module manages and multiplexes the accesses to peripheral devices from the application tasks.

We subdivide our operating system into seven individual modules: boot code, process management, hardware abstraction layer, scheduler, inter-process communication, memory manager, resource manager. Our design defines the interfaces of all modules and provides fixed implementations of two of them: the boot code and the process management. In combination with the module interface specification, we call these two modules the core functionality of our operating system. Except for this small part of the kernel, the whole operating system is designed to be configurable and adaptable. However, to build a fully functional operating system three more modules are required: the hardware abstraction layer, the scheduler, and the inter-process communication module. We call these modules the essential modules. The aforementioned six modules can be complemented by other optional modules: the memory manager or the resources manager module. A special feature of the optional modules is that they run on top of the abstraction layer defined by MUC-OS and thus do not notice or care on which processor core they are actually executed. An overview of the module structure of MUC-OS and the functions of the individual modules is given in Table I.

We have chosen the presented module structure to cover most functions of a typical operating system, as described in well-known textbooks [*3*] [*4*] [*5*]. In case a specific use case requires additional modules, those can be easily integrated into the system. However, for most embedded application scenarios it should be sufficient to adapt the existing modules and implement an application specific version. We tried to reduce the efforts required to maintain or extend our operating system, as well as the initial training hurdle, by avoiding a strict component based programming model and by providing simple, well documented module interfaces. Our goal is to provide a platform for the evaluation of future operating system concepts or components that could be easily used and extended by students. As a consequence, we specified the core functionality of our operating system as fixed, as mentioned above.

Configurable operating systems are not a new approach; several competing concepts exists, so we briefly explain how our approach fits into the research landscape of configurable operating systems: Research lists more than 15 different configurable operating systems [*6*] [*7*], out of which four suggest comparable approaches: eCos [*8*], MMLite [*9*], Pebble [*10*], Think [*11*], and Exokernel [*12*]. However, in contrast to MUC-OS, eCos stops the configuration and modularization at the kernel level, while our approach allows to flexibly adapt the kernel to the very specific needs of an application. MMLite and Think address embedded system and systems with restricted resources in general; however they focus explicitly on the support of dynamic re-configurations of the final system and thus suffer certain performance degradations. With MUC-OS, we lay our focus on static systems and compile-time configuration of the whole operating system to deliver the best possible performance of the system. Pebble also focuses on embedded applications and reduces the fixed function set of the operating system to a minimum, as we do. However, they restrict all functions outside this small “kernel nucleus” to the user space, hence inducing a potentially high processing overhead due to frequent context switches. The Exokernel project suggests a

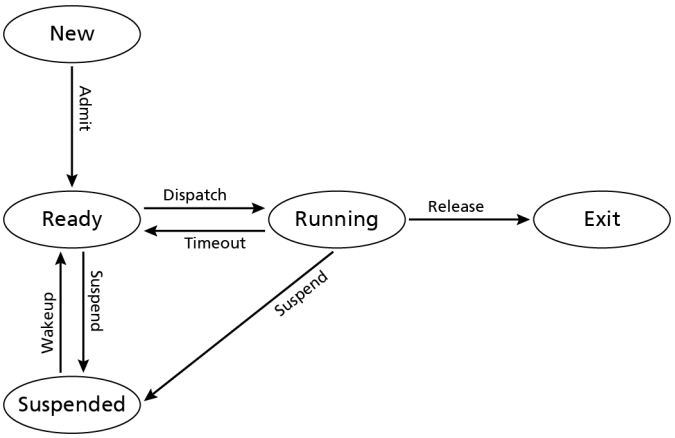

Fig. 2. Extended five-state task model of MUC-OS, based on the original work by Stallings [5].

more competitive approach by utilizing static libraries to construct the overall operating system. Accordingly, Exokernel is classified as a so called library operating system. However, Exokernel does not target resource restricted embedded systems, but focuses on large distributed systems.

In Fig. 3 we set the attributes of MUC-OS in relation to those of the four configurable operating systems for embedded systems: eCos, MMLite, Pebble, and Think. The original classification of the four operating systems and the illustrative diagram were taken from [*7*]. We complemented the original diagram with the attributes of our proposal according to the guidelines given by [*7*]. The axes of the diagram shall be interpreted as follows:

- G (granularity) quantifies the size of the smallest configuration unit defined by the examined operating system.
- D (depth) quantifies the size of the smallest non-configurable part of an operating system. An operating system is either fully configurable (*full*) or has a fixed set of functions that cannot be configured. Depending on the size of the fixed part, an operating system is either identified as partially-small (*p-small*), -medium (*p-med*), or -big (*p-big*). Where *p-small* stands for a rather small fixed function set and *p-big* for an almost non-configurable operating system with a large fixed function set.
- T (time) quantifies the point during the life cycle of an operating system until which the system can be configured: at development time only (*dvpt*), at boot time (*boot*), or even at runtime (*execution*).
- I (integrity) examines the possibility to ensure the validity of a new operating system configuration; the options are: *none*, *semantic*, *security*. The *semantic* integrity can be achieved through interface type-checking, a dedicated language, constraint definitions, or the framework itself. The *security* level reflects system that actively ensure the correctness of any configuration, e.g. by protection domains or configuration policies. It should be noted that every operating system that falls into the last category also provides a semantic validation.

For a more detailed discussion of the attributes and the classification of the individual operating systems we kindly refer the interested reader to the original work [*7*].

We provide a fully functional prototype of MUC-OS that implements the core functionalities and all essentials modules. To evaluate the expandability of our flexible operating system design, we applied the MUC-OS prototype to two case studies, which each required their own implementations of the scheduler and inter-process communication modules. These two case studies and their impact on the module implementations are discussed in the following section.

IV. Application Examples

In this section we present two example configurations of our operating system platform. Each of the scenarios presents itself with different requirements, making it necessary to use different implementations of the scheduling and inter-process communication modules.

A. VoIP Processing Platform

The first example is a voice processing application from the field of telecommunication. The specific requirements of this case study include (soft) real-time constraints, given in terms of a number of simultaneous voice channels that need to be processed without loss of data. Further constraints are presented by the target system, which is a homogeneous multi-core platform. To reduce fabrication cost and energy consumption, the design does not have hardware support for cache coherence, so special care has to be taken when accessing memory. We configured our operating systems with three objectives: 1. provide a consistent environment to the VoIP application despite the absence of coherent caches, 2. fully utilize the available processing power by dynamically managing task distribution among the cores, and 3. reduce energy consumption by switching off cores in times of low system utilization.

As the timing requirements of the application focus on throughput rather than guaranteed compliance with a deterministic task schedule, we configured the system to use a preemptive round-robin scheduler and a queue-based IPC mechanism that provides best-effort message delivery. To address the absence of coherent caches, we specifically adapted the IPC mechanism and the implementation of the message queues. We also provide a basic memory management module to execute required operations whenever regions of memory are accessed by more than one core. On top of these basic functions, we provide a service module for global load management. It can either balance load evenly, thus optimizing throughput, or optimize for low energy consumption by concentrating tasks on few cores, making it possible to switch off other cores. Detailed information on our load management module is provided in [*13*] and [*14*].

B. Automotive ECU

The second example is an automotive control unit for a small electric vehicle. Due to price and energy constraints, the electric vehicle has to implement all its software functions on only one electronic control unit (ECU). This ECU has to man-

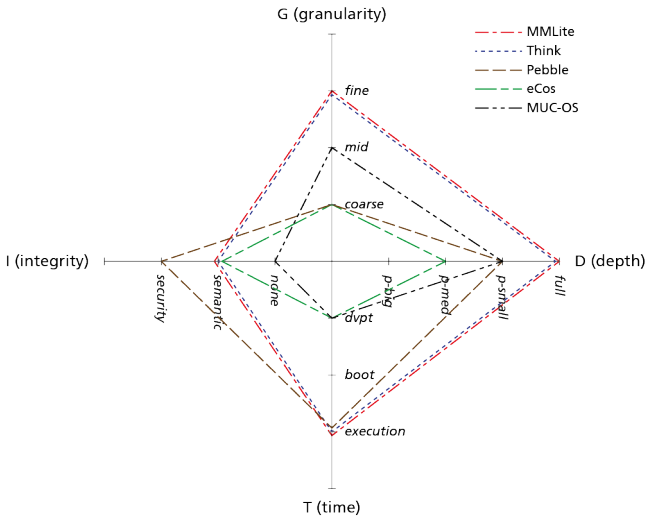

Fig. 3. Comparison of MUC-OS against existing research in the area of configurable operating systems, based on the on the results of [*7*].

age the control of the engines and breaks, as well as the infotainment system. Accordingly, the operating system on that ECU has provide a deterministic environment, easily accessible by runtime analysis, while at the same time being able to execute tasks of different criticality in parallel. Again, we configured our operating system with three objectives: 1. provide an execution environment that allows a dependable execution of applications with hard real-time constraints, 2. Implement trustworthy isolation of tasks of different criticality, and 3. minimize the energy consumption by relocating tasks and switching off cores in times of low system utilization.

Our configuration reflects the objectives as follows: As scheduler, we chose a rate monotonic scheduler. The rate monotonic scheduler allows to specify a second scheduler module that handles the scheduling of best-effort task in free time slots within the schedule plan. We have configured a simple round robin scheduler with time slices for that purpose. With the combination of a rate monotonic and a round robin scheduler we can guarantee a temporal isolation of the different tasks and dependable issue time for hard real-time applications. To completely fulfill objectives 1 and 2, however, further steps are needed. A dependable execution of hard real-time tasks requires an analyzable execution environment and a trustworthy isolation of mixed criticality tasks not only requires a temporal isolation, but also a spatial isolation. Thus, we configured the system with an IPC module that supports a deterministic message delivery and with a memory management module that provides memory protection mechanisms to isolate different tasks in memory. In addition to the mentioned modules we reused the global load management service from the first example to optimize the system for low energy consumption. More details on the configured information and communications technology (ICT) platform for electric vehicles and the deterministic IPC mechanism are presented in [*15*].

V. Conclusion and Outlook

This paper gives an overview on the standard software deployment strategies for multi-core platforms: SMP and AMP. We postulate that neither of these approaches is a likely fit for embedded systems in general. Thus, we purpose a new combined deployment strategy which utilizes the AMP concept with a message based communication infrastructure that provides a distributed execution environment, like SMP systems do. We embed the new deployment strategy into a flexible operating system called MUC-OS. The MUC-OS kernel can be adapted to specific platforms and applications by configuration and implementation of five individual modules. We evaluated the MUC-OS design within two example configuration: a VoIP Processing Platform and an Automotive ECU.

In future, we plan to extend MUC-OS with basic virtualization capabilities in form of a hierarchical scheduler, a temporal and spatial isolation of tasks, and a real-time capable peripheral virtualization.

VI. References

[1] A. Baumann, et al., "The Multikernel: A New OS Architecture for Scalable Multicore Systems," in *Proc. of the 22nd Symp. on Operating Systems Principles*, New York, NY, USA, 2009, pp. 29-44.

[2] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt, "Helios: Heterogeneous Multiprocessing with Satellite Kernels," in *Proc. of the 22nd Symp. on Operating Systems Principles*, New York, NY, USA, 2009, pp. 221-234.

[3] A. S. Tanenbaum, *Modern operating systems*, 2nd ed. Upper Saddle River, NJ: Prentice-Hall, 2001.

[4] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating system concepts*, 7th ed. Hoboken, NJ: Wiley, 2005.

[5] W. Stallings, *Operating Systems: Internals and Design Principles*, 4th ed. Pearson/Prentice Hall, 2000.

[6] L. F. Friedrich, J. Stankovic, M. Humphrey, M. Marley, and J. J. Haskins, "A survey of configurable, component-based operating systems for embedded applications," *IEEE Micro*, vol. 21, no. 3, pp. 54-68, 2001.

[7] J.-C. Tournier, "A survey of configurable operating systems," University of New Mexico, Albuquerque, NM, USA, Tech. rep., 2005.

[8] eCos Website. [Online]. http://ecos.sourceware.org/

[9] J. Helander and A. Forin, "MMLite: A Highly Componentized System Architecture," in *Proc. of the 8th European Workshop on Support for Composing Distributed Applicat.*, New York, NY, USA, 1998, pp. 96-103.

[10] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz, "The Pebble Component-Based Operating System," in *Proc. of the USENIX Annu. Tech. Conf.*, 1999.

[11] J.-P. Fassino, J.-B. Stefani, J.-B. Stefani, and G. Muller, "Think: A Software Framework for Component-based Operating System Kernels," in *Proc. of the Annu. USENIX Conf.*, 2002.

[12] D. R. Engler, M. F. Kaashoek, and J. ,. J. O'Toole, "Exokernel: An Operating System Architecture for Application-level Resource Management," *SIGOPS Oper. Syst. Rev.*, vol. 29, no. 5, pp. 251-266, Dec. 1995.

[13] M. Becker, "Evaluation of Load Balancing Methods on Embedded Multicore Systems and their Effect on Power Consumption," Master's thesis, University of Applied Sciences Munich, 2013.

[14] M. Becker, A. Schmidt, and M. Orehek, "Saving Energy by Means of Dynamic Load Management in Embedded Multicore Systems," unpublished.

[15] O. Horst, P. Heinrich, and F. Langer, "ICT-Architecture for Multimodal Electric Vehicles," in press.

[16] J. Holt, et al., "Software Standards for the Multicore Era," *IEEE Micro*, vol. 29, no. 3, pp. 40-51, 2009.