Software Plattform Embedded Systems 2020

GEFÖRDERT VOM

Bundesministerium
für Bildung
und Forschung

- Deliverable D5.2.A: Prerequisites for model-based deployment -

Version: 1.1

Projektbezeichnung	SPES 2020	
Verantwortlich	Robert Hilbrich, Hartmut Lackner	
QS-Verantwortlich	<QS-Verantwortliche Organisation>	
Erstellt am	21.10.2009	
Zuletzt geändert	23.06.2010 11:04	
Freigabestatus		Vertraulich für Partner: <Partner1>; <Partner2>; …
		Projektöffentlich
	X	Öffentlich
Bearbeitungszustand		in Bearbeitung
		vorgelegt
	X	fertig gestellt

Weitere Produktinformationen

Erzeugung	<Autor des Dokuments>
Mitwirkend	<Co-Autoren des Dokuments>

Änderungsverzeichnis

Änderung			Geänderte Kapitel	Beschreibung der Änderung	Autor	Zustand
Nr.	Datum	Version				
1	21.10.09	0.1	Alle	Initiale Produkterstellung		
2	24.02.10	1.1	Alle	Einarbeitung QS		

Abstract

This deliverable starts with noting that embedded systems designers are interested in applying multi-core architectures in their designs in order to increase processing power. Legacy software however is often not suited to be executed on such platforms. Software migration and design can be simplified by applying model-based development, which abstracts from hardware and thus improves the portability of code. However multi-core architectures introduce new parameters that might affect the requirements, for instance the timing constraints. Several methods for deploying models on multi-core architectures are discussed in this document. Then prerequisites for model-based deployment are presented which concern the application, the operating system, and hardware dependent software.

Inhalt

1. The need for model-based deployment

1.1. Current practice in embedded system design

Embedded real-time systems as can be found in the automotive and aerospace domains nowadays are often complex distributed systems, that is, multiple Electronic Control Units (ECUs) connected through a bus or network (1) , (2). Low-level software development for these systems is cumbersome and results in code that is not portable and thus difficult to reuse. To cope with the complexity and increase portability, software developers have started to use models that represent the system behaviour but abstract from low level details (3). Later in the design process these models can be deployed on a specific target platform.

1.2. The trend towards multi-cores

In desktop and high-performance computing clearly a trend towards multi-core processors is visible (4). The term multi-core is used here to indicate that there are multiple processing cores on a single chip. In the context of SPES 2020 several industrial partners have showed interest in the use of multi-cores in embedded systems. Two example cases provided by these partners will be presented here to motivate the research in this area.

- Für die Sicherheit im Luftfahrtbereich zählen die Einsetzung von Funktionen und Verteilung der Rolle der Geräte in der Systemarchitektur zu den wichtigsten Kriterien. Auβer der Steigerung der Performanz soll die Einsetzung von Multicore in Flugzeugen eine Reduktion der Anzahl von Geräten ermöglichen, oder anders formuliert, es sollte eine Konzentration der Funktionen auf Weniger Geräte bewirken. Aspekte wie Redundanz, Dissimilarität und Determinismus (Vorhersehbarkeit) werden mit den aktuellen Single-Core System-Architekturen beherrscht, und zählen als Basis für die Sicherheitsberechnung. Die Einsetzung von Multicore, mit zahlreichen vereilten bzw. gemeinsamen Ressourcen von der Hardware (Cache, MMU, Speicher, Bus, etc.) bringt zahlreiche neuen Parameter mit noch unbekannten Konsequenzen für die Sicherheitsaspekte.
  Source: *Beschreibung der Fallstudie im Anwendungsgebiet Avionik v0.8 , 3.10 Fallbeispiel: Luftfahrtsicherheitsaspekte bei Einsetzung von Multicore Technologie,*
- In dieser Task geht es darum, ein medizinisches Handhabungsgerät für diagnostische Röntgenaufnahmen als Beispiel für Robotik in der Medizin (weiter) zu entwickeln. Ziel ist ein sicheres Abfahren von Trajektorien im Raum durch mehrachsige mechanische Trägersysteme für Röntgenanlagen. Hierzu ist eine Pfadplanungskomponente von einem Ausgangspunkt im Raum zu einem Zielpunkt zu entwickeln, unter Berücksichtigung von statischen und dynamischen Sicherheitszonen (Wände, Untersuchungstisch, Patient auf dem Untersuchungstisch). Es ist zu untersuchen, in wie weit eine Kopplung mit optischen oder Laser-basierten Überwachungssystemen zur Erkennung von Hindernissen (Laborwagen, Personal usw.) erfolgen kann bzw. muss. Bei der Entwicklung des Demonstrators sind weitere Sicherheitsanforderungen für die Zulassung zu berücksichtigen.

  Als erstes ist eine Modellierung der Sicherheitszonen durch Kugelradien in einem geeigneten Modellierungsformalismus vorzunehmen (siehe ZP-AP 1). Sodann erfolgt die Realisierung eines Pfadplaners für einen Industrieroboter, der als Träger-

> system für Röntgenröhre und Röntgendetektor dient. Dabei sind die Ausweich-Räume, die für die einzelnen Achsen notwendig sind, entsprechend zu berücksichtigen. Als Verfahren soll hier untersucht werden, in wie weit eine regelbasierte Optimierung der Trajektorie möglich und effizient realisierbar ist. Weiterhin ist die Anwendbarkeit der Potenzialfeldmethode zur Konfliktauflösung bei Annäherung an die Sicherheitszone zu berücksichtigen. Es ist ein Anti-windup Algorithmus zur Berücksichtigung der Kabelführung bei Bewegung zu konzipieren und umzusetzen. Als nächstes ist eine Methode zur mehrfachen Abfahrt der selben Trajektorie (Reproduzierbarkeit, mechanisches Spiel) algorithmsch zu formulieren und prototypisch zu realisieren. Abschliessend erfolgt eine Optimierung des Pfadplanungs-algorithmus auf multi-core CPUs, um die Echtzeitanforderungen zu erfüllen (siehe ZP-AP 5).
> Source: *MT-Task 3.1 Kinematik mehrachsiger bewegter Systeme in einem Röntgenuntersuchungsraum*

Generally speaking embedded systems in the automotive and avionics domains could benefit from the use of multi-core processors because the processing power of a single ECU is increased. This means more jobs can be hosted which results in a reduction of hardware and thus of cabling contact points (1). Also the power consumption versus performance ratio is usually better than in single-core architectures (2). On the other hand it can be difficult to fully utilize the hardware capabilities, for instance because of inherently sequential code and non-optimal load-balancing.

Although multi-core platforms suitable to be used in embedded systems are already available, e.g. systems on chip, embedded systems designers have not yet switched to these platforms on a massive scale. It is argued in (5) that this is because of the huge development investments in existing software which is usually designed for single core platforms. A complete redesign of these systems to allow efficient execution on a certain multi-core platform would be very costly. To cope with this problem model-based development can be used, similar to the current practice in distributed embedded systems.

The remainder of this deliverable is structured as follows: the next section elaborates on the modelling of software components. Then in the third section concepts for the deployment of software on multi-core platforms are introduced. In the fourth section the prerequisites for model-based deployment are derived based on the previous sections. The fifth section then presents related work in this area and the sixth ends with the conclusion.

2. Modelling of software components

Models can be extracted from existing software components in order to obtain a representation of the behaviour of a system that is abstracted from details about the underlying hardware. The obtained high-level models can be connected together or with new components and then be used to simulate and verify the behaviour of the application. Then tools can be used to deploy the application on a certain hardware platform, for instance by means of automatic code generation. This approach makes the migration of software relatively easy and enables the re-use of software (3). New components can be developed in a model-based fashion which frees the designer of low-level details and generally speeds up the design process. In industry Simulink is often used for the model-based design of software.

In model-based software development it is important to correctly implement the requirements and constraints that apply to the application. In embedded real-time systems this includes the overall functionality, time restrictions, resource limitations, and scheduling (6). The designer must know which of these aspects must be implemented in the model and which will be dealt with at a later stage. In (7) for instance, the introduction of explicit communication delays early in the development process is shown to help minimizing the effort of application deployment.

Other reasons to introduce hardware specific parameters early in the design process can be thought of: timing requirements are critical for the correct functioning of real-time systems and they are almost always dependent on the underlying hardware. This means that the model must have some notion of the target architecture at some point in the design process in other to be able to implement and verify those requirements.

In (7) three abstraction levels are proposed. These are presented here to illustrate the concept of hardware abstraction. On the highest level the structure and functional dependencies are captured, e.g. by using system structure diagrams or UML. Going more into detail, the next lower level presents the dataflow, communication and behaviour of the different components, e.g. using finite-state machines and low-level dataflow diagrams. On this level validation and simulation can be performed. The third and lowest level contains unique details of the target platform. This is consistent with the two perspectives in (5), namely a platform independent view that captures the functional behaviour similar to the first two levels described above, and a platform dependent view that provides a mapping between the abstract models and the actual hardware.

Because multi-cores have a *multiple instruction streams, multiple data streams* (MIMD) architecture they can exploit thread level parallelism. To take advantage of an architecture with *n* cores there must usually be at least *n* threads to execute. These independent threads should either already be separated in the model or a tool should be able to extract them in a later stage. The threads can be large, independent processes but can also be small pieces of code, e.g. a loop. The amount of computation that is done in a loop is called the *grain size* and is an important parameter in considering how to exploit the thread-level parallelism efficiently (4).

3. Deployment on multi-core architectures

3.1. Concepts in multi-core computing

Parallel computing has been done for a long time, for instance in distributed computing. Here, computational nodes are connected through a bus or network to form a machine capable of executing multiple instructions in parallel. There are many similarities between those parallel computers and modern multi-core processors, so it is no surprise that some ideas were adapted. The main difference with recent multi-cores however is that those generally have on-chip networks that allow fast and dependable communication. Furthermore usually some resources like memory, caches and I/O are shared between the cores, which is not common in distributed computing (4).

There are two models for data exchange between computational nodes in parallel computing: message passing and accessing a shared address space. In the former cores communicate explicitly by sending messages; they have no access to each others memory. In the latter all memory is accessible to the programmer wherever it is located. Each core can have its own physical memory in which case the memory access time is non-uniform (NUMA architecture), or the memory can be centralized (8). In the latter case the cores can have separate caches or access a common cache, or a combination of those two if there are more levels of cache. When a centralized memory is used it is generally considered infeasible to scale multi-core systems beyond a few dozen processors.

Another distinction can be made: in symmetric multiprocessing (SMP) a common Operating System (OS) manages all cores while in asymmetric multiprocessing (AMP) each core is managed by a separate (or a separate copy of the same) OS. Whereas SMP allows migration of tasks between cores to balance the workload, AMP offers an execution environment similar to that of a single-core processor which facilitates the use of legacy software. The main disadvantage of AMP is that load balancing is difficult because processes cannot be migrated between cores. SMP on the other hand offers a true concurrent environment in which it is often difficult to run legacy code. Bound multiprocessing combines the advantages of both: a single OS is used but tasks can be locked to cores (5) , (2).

The term multi-core is nowadays used for different classes of processors. In the PC market it often refers to homogeneous arrays of processors that are located on the same chip, share memory and perform SMP. When talking about embedded systems the term multi-core often refers to Systems on Chip (SoC) that feature multiple processor cores (9). These might also be shared-memory and perform SMP but any combination is possible; a SoC that contains both a general core and a DSP core for instance can perform AMP and have private memories.

3.2. Approaches to deployment

A common approach in modern embedded application development is the use of an OS, the alternative being the development of software from scratch. The obvious advantage of using an OS or Real-Time OS is the availability of high-level functionalities while the designer is still able to define certain functions as firmware and compile them for specific cores. However, this approach can be too generic and result in poor performance and memory overhead. If the functionality offered by an OS is not really needed it might be more cost-effective to develop the complete application specifically for the target platform. This approach is however not suitable for large and complex applications and lacks portability (10).

From the perspective of model-based deployment the use of an OS is possible but not mandatory. The different strategies that have been proposed to deal with the problems of both approaches are described in this sub-section. It should be noted that some solutions propose the use of certain additional software which increases the trusted code base and can thus have major implications on certified software.

If an OS is not really needed, the problems of complexity and lack of portability can be overcome if high-level models are used for *automatic code generation* (11). Critical sections can still be hand coded to guarantee that certain requirements are met or to increase performance.

Another way of deploying high-level components on multi-core platforms is the use of hardware dependent software (HdS), for instance a *Hardware Abstraction Layer* (HAL). Such a layer provides an API that allows access to the hardware but hides platform specific issues (12). HdS does normally not provide high level services like scheduling and memory management, but rather low level functions like initializing hardware, booting the system and access to debugging facilities. The idea of hardware dependent software is depicted in figure 1, more details are given in the next sub-section.

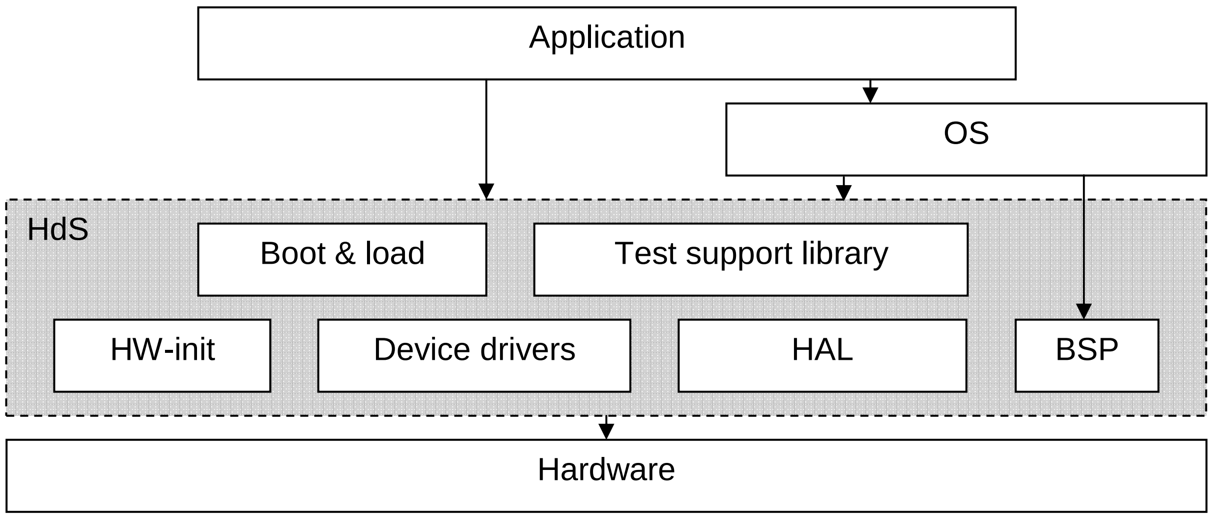

Figure 1: Hardware dependent Software

A *device driver* is another abstraction method, usually the term is used for a hardware-dependent component that allows higher-level software to access a certain device. It is then similar to a HAL.

If higher level functionalities are needed an operating system can be employed. However, for modern heterogeneous multi-core systems a tailored OS that fits the needs is not always available. In that case again hardware dependent software can be used. If a HAL is OS dependent it is often referred to as a *Board Support Package* (BSP).

As was stated before, an OS might use too much hardware resources and thus result in poor performance and memory overhead. A solution to this could be the use of a *component-based operating system*. When an OS is divided in components that each provide a set of functionalities, an instance of the OS with a minimal footprint can be generated to fit the needs of a certain application (13) , (10).

Microkernels (µ-kernels) are another approach to minimizing the OS footprint. Here the mechanisms contained in the OS (kernel functions) are spread over small, separated processes that interact through inter-process communication: the µ-kernels. In the so-called second generation µ-kernels all protocol-related functions are implemented as servers that can be started and stopped at will. This leaves only the µ-kernels that contain critical mechanisms like basic scheduling and memory management continuously running. This approach is considered robust because crashed processes can simply be stopped and started again. Third-generation µ-kernels are characterized by security-oriented APIs. Recently complete formal verification of the embedded OS seL4 has been achieved (14). Similar to operating systems, a number of µ-kernels have to be rewritten for each target processor. The modular nature of the µ-kernels approach however simplifies porting the OS because the functionality is clearly separated. The term *nanokernel* may refer to particularly small µ-kernels but is also used as synonym for HALs.

Virtualization can also be used for the deployment of software on embedded multi-core systems (15). A Virtual Machine (VM) is an isolated software duplicate of a machine. It executes on a real machine and is usually managed by a hypervisor which can schedule multiple VMs to run at the same time on the same hardware. Advantages of this approach are that the complete functionality of a system is available to the application, but a fault occurring in one VM cannot spread through the system. There is however overhead like the management of the VMs and inter-VM communication. A VM running on a part of the system has no knowledge of the global system architecture. If high-level (legacy) code is compiled for such a machine it runs in a protected environment and communicates with the other sub-systems on the chip as if they were different machines. This is similar to the approach of asymmetric multi-processing with the difference that multiple VMs can share a sub-system, thus partly solving the problem of load-balancing. VMs are often however not very efficient, for instance due to context switches and the resource usage of the hypervisor.

3.3. Hardware dependent Software

It has become clear that in the design of software targeted for multi-cores in general and multiprocessor SoCs (MPSoC) in particular it is useful to abstract from hardware to be able to port code between different platforms. This section will focus on one of the ways in which this abstraction can be achieved, namely with hardware dependent software.

Architectural properties that are important when implementing software on multi-core platforms are for instance the number of cores, the communication network architecture and the memory hierarchy. Many more exist and differ per platform. Besides these there are properties that apply to a certain sub-system, for instance a core. Properties for such a sub-system can be the type of the core, the execution context, interrupt and memory management, but again many others can be thought of. All these properties define the system and should be addressed in the HdS in order to allow the application to efficiently use the hardware.

Hardware dependent Software describes the software that is directly dependent on the underlying hardware, and is accessible through an API that allows the application to access the hardware without addressing specific details. The API should provide an abstraction of for instance data types, task context, interrupt management, I/O, and memory management. This means functions embedded in the HdS could include booting the system, performing context switches, and the configuration of and access to hardware resources. For each different hardware platform at least a part of the HdS needs to be rewritten. If the API is not changed however, the application is portable between the different architectures. Another advantage that HdS offers is the early validation of software. If the API is designed in an early stage, software validation can already be performed and the overall design cycle can be reduced (16). Most forms of HdS that currently exist are board support packages which are OS dependent.

Another approach which is presented in (17) starts from an application model consisting of processes that communicate via abstract message passing channels. When this model is mapped to a platform net list that contains a description of the embedded platform, it is possible to generate a Transaction Level Model (TLM). This TLM can be used for the validation of communication between the processes mapped to the different cores. This enables early application development. After the HdS is developed it can be used together with the OS to refine the TLM into a Pin-Cycle Accurate Models (PCAM). The PCAM allows an accurate estimation of performance, design cost and power consumption. Because the manual development of HdS can be complex and time consuming this approach uses automated HdS synthesis, the set of design parameters needed for this can be extracted from the application and the platform.

3.4. Timing analysis

In time-critical embedded systems the correctness of a system depends on the results of the computations as well as the time at which those results become available. If those systems are safety-critical, reliable guarantees for the satisfaction of the timing constraints are needed. Currently static timing analysis is often used to get such guarantees, for instance in the automotive and aerospace domains. Tools that perform such analysis use abstract architectural models and can handle complex architectures for which the execution time can greatly vary, e.g. due to speculative execution. Some architectures cannot be analyzed at all, e.g. when caches with a random-replacement strategy are used. Most of the problems in static timing analysis are caused by the interference on shared resources. A situation in which two entities try to access the same resource can only be modelled if the timing behaviour of all related components is deterministic. Because resources are often shared on multi-core platforms it is clear that static timing analysis can be difficult. Shared caches and communication networks in particular can make timing analysis infeasible because of non-deterministic behaviour. To obtain predictable timing behaviour in a multi-core platform the interference on shared resources should be eliminated wherever possible (18). Besides the hardware-specific issues the timing analysis should also consider the software layers on top of that, for instance the HdS. These layers should not introduce any non-deterministic behaviour. It could even be considered to design them to simplify timing analysis, for instance by cleaning the cache at certain predefined points.

4. Prerequisites

From the previous sections some prerequisites for the model-based deployment of software on multi-core platforms can be derived. The prerequisites are divided into three groups: application related, OS related and HdS related. Note that not all of these prerequisites apply to each different strategy for model-based deployment, they are a set of which some apply for each situation.

1. Application related prerequisites:

   1.1. A high-level model of the application should be available.
   1.2. The model should be abstracted from the target hardware as much as possible but this goal should not interfere with the implementation of requirements and constraints.
   1.3. The application should contain thread-level parallelism. The optimal grain size should be determined by considering the target platform.

1.4. All requirements and constraints that are necessary for the correct functioning of the application should be implemented in the model.

2. Operating system related prerequisites:

2.1. A choice should be made between asymmetric multiprocessing, symmetric multi-processing or bound multiprocessing.
2.2. The OS should provide an interface to the application that is abstracted from the underlying hardware architecture.
2.3. When the OS is not tailored for a specific platform it should access the hardware through a layer of hardware dependent software.

3. Prerequisites for the hardware dependent software:

3.1. HdS should provide an API that allows application designers to access the hardware in a convenient way.
3.2. HdS should hide hardware specific issues from the software on top of it.
3.3. HdS should allow the application to fully utilize the target platform.
3.4. For real-time applications the behaviour of the HdS must be deterministic.

5. Related work

In (17) the model-based design of an MP3-decoder deployed on an MP-SoC is described. The Embedded System Environment tool set was used which enables transaction level design of multi-core systems. The HdS synthesis in this tool is automated and provides a library of application level services that allow access to the hardware. The application was implemented on a heterogeneous multi-core SoC consisting of a software core and four hardware cores. The automatic HdS synthesis reduced overall design time.

The Multicore Association is an organization that develops software standards that address the new challenges posed by embedded multi-core systems. In (19) the Multicore Communications API (MCAPI) is described which aims to standardize small-footprint and efficient inter-core communications. It starts with the observation that it is impractical to work with communication standards designed for SMP or distributed computing when dealing with heterogeneous multi-cores. There are alternative standards but those too are often not suitable because they cannot deal with heterogeneity of hard- and software and constraints on code size and execution time overhead. This means new communication standards have to be developed in order to simplify the software development for multi-core systems and improve the performance. The API that is presented targets portability and scalability, and represents an interesting example of hardware abstraction. It allows the designer to choose different types of communication methods depending on the context. The MCAPI performs quite well and can be seen as a foundation for higher levels of functionality.

In (20) a high-level component-based design approach for multi-core SoCs is presented. Because SoCs become more and more complex the authors identify the need for new design methods because:

- Communication protocols cannot be verified at register-transfer level because it is too time consuming, thus higher abstraction levels are needed

- Heterogeneous multi-core SoCs will include processors with different instruction set architectures so the software complexity will increase and cannot be programmed in low-level languages
- Hardware/software interfaces are required to implement an application-specific communication interconnection

In this paper the system is represented by a virtual components interconnected via channels. These components consist of a wrapper and an internal component, the wrapper handles all accesses to the external channels and is generated automatically by the design environment. Those wrapper-generation tools are able to synthesize hardware interfaces, device drivers, and operating systems that implement a high-level interconnect API. The design of a VDSL modem shows that the high-level component-based approach reduces the design effort to four months, compared to an estimated five person year effort for manual design.

In (10) a component-based system framework is presented that provides high-level system services to embedded software applications on heterogeneous SoC, while minimizing memory usage and any negative impacts on the performance. The framework is composed of a hardware abstraction component and the DNA Operating System, which are both designed for portability and modularity. The hardware abstraction component provides access to hardware-specific primitives. It handles for instance differences in endianness, inter-processor synchronization and execution context. The DNA-OS is component based and similar to the µ-kernels. Its components are generic and use the hardware abstraction to access the hardware. The system framework is successfully used in two projects: SoCLib and SHAPES.

6. Conclusion

This deliverable started with an analysis of current practice in the field of embedded systems.
It became clear that multi-core architectures are not yet widely adopted in this market. This is partly due to the investments in current software which is often targeted for single-core systems. Due to the potential advantages however it can be expected that future embedded processors will feature multi-cores, following the trend in desktop-computing.

Manual low-level software design for such architectures is complex and results in code that is not portable. These problems can be coped with by the model-based development of software: applications are abstracted from the target hardware so that they are portable. The implementation of requirements and constraints may sometimes conflict with the desired abstraction. An example of this are real-time systems, where the correctness of the system depends on timing requirements which in turn depend on the underlying hardware. Furthermore the hardware itself introduces new parameters of which the designer should be aware, cores might for instance share resources via a network which can lead to timing behaviour that is difficult to predict.

There are different approaches for the deployment of models on a target platform. One is to use tools to automatically generate code specifically for the target platform. Another is to have a layer of Hardware dependent Software that separates the application from the hardware. Such a layer offers a fixed API to the application while the underlying hardware dependent code can be changed to fit a specific platform. Yet another approach is to use an OS which offers high-level functionalities to the application. Here again HdS can be used to obtain portability. Operating systems can be component based so that a minimal instance of an OS can be generated for a specific project. This is similar to the use of an OS based on µ-kernels. The use of virtual machines offers separated environments within a system which prevents faults from spreading and facilitates re-use of legacy software, which however increases the usage of resources.

In section four prerequisites for the model-based deployment of software on multicore architectures are presented. These prerequisites can be used as a start for further research on model-based deployment.

7 Bibliography

1. **Kopetz H., Obermaisser R., El Salloum C., Huber B.** *Automotive Software Development for a Multi-Core System-on-a-Chip.* s.l. : Vienna University of Technology, 2007.
2. **Leroux P., Craig R.** *Migrating legacy applications to multicore processors.* s.l. : QNX Software systems, 2006.
3. **Kim J. E., Kapoor R., Herrmann M., Haerdtlein J., Grzeschniok F., Lutz P.** *Software Behaviour Description of Real-Time Embedded Systems in Component Based Software Development.* s.l. : Robert Bosch, 2008.
4. **Hennessy J. L., Patterson D. A.** *Computer Architecture: A Quantitative Approach, 4th edition.* s.l. : Elsevier & Morgan Kauffmann, 2007.
5. **Nemati F., Kraft J., Nolte T.** *Towards Migrating Legacy Real-Time Systems to Multi-Core Platforms.* s.l. : Mälardalen Real-Time Research Centre, 2008.
6. **Schätz B., Broy M., Huber F., Philipps J., Prenninger W., Pretschner A., Rumpe B.** *Model-Based Software and Systems Development.* s.l. : TU München, ETH Zurich & TU Braunschweig, 2004.
7. **Bauer A., Romberg J.** *Model-Based Deployment: From a High-Level View to Low-Level Implementations.* s.l. : Technische Universität München, 2004.
8. **Grama A., Gupta A., Karypis G., Kumar V.** *Introduction to Parallel Computing, 2nd Edition.* s.l. : Addison Wesley, 2003.
9. **Leibson, S.** *Multicore microprocessors and embedded multicore SOCs have very different needs.* s.l. : Tensilica, Inc., 2007.
10. **Guerin X., Petrot F.** *A System Framework for the Design of Embedded Software Targeting Heterogeneous Multi-Core SoCs.* s.l. : TIMA Laboratory, 2009.
11. **Edwards, C.** *Model Development.* s.l. : IEEE Review, 2003.
12. **Yoo S., Jerraya A. A.** *Introduction to Hardware Abstraction Layers for SoC.* s.l. : TIMA Laboratory, 2003.
13. **Massa, A. J.** *Embedded Software Development with eCos.* s.l. : Prentice Hall, 2003.
14. **Klein G., Elphinstone K., Heiser G., Andronick J., Cock D., Derrin P., Elkaduwe D., Engelhardt K., Kolanski R., Norrish M., Sewell T., Tuch H., Winwood S.** *seL4 - Formal Verification of an OS Kernel.* 2009.
15. **Kaiser, R.** *Complex Embedded Systems - A Case for Virtualization.* s.l. : Wiesbaden University of Applied Sciences, 2009.
16. **Yoo S., Youssef M. W., Bouchhima A., Jerraya A. A., Diaz Nava M.** *Multi-Processor SoC Design Methodology using a Concept of Two-Layer Hardware-dependent Software.* s.l. : TIMA Laboratory & ST Microelectronics, 2004.
17. **Abdi S., Schirner G., Viskic I., Cho H., Hwang Y., Yu L., Gajski D.** *Hardware dependent Software Synthesis for Many-Core Embedded Systems.* s.l. : University of California, 2009.
18. **Wilhelm R., Grund D., Reineke J., Schlickling M., Pister M., Ferdinand C.** *Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems.* s.l. : Saarland University & AbsInt Angewandte Informatik GmbH, 2009.
19. **Holt J., Agarwal A., Brehmer S., Domeika M., Griffin P., Schirrmeister F.** *Software Standards for the Multicore Era.* s.l. : The Multicore Association, 2009.
20. **Cesário W., Baghdadi A., Gauthier L., Lyonnard D., Nicolescu G., Paviot Y., Yoo S., Jerraya A.A., Diaz-Nava M.** *Component-Based Design Approach for Multicore SoCs.* s.l. : TIMA Laboratory & ST Microelectronics, 2002.