Introduction to Hardware-dependent Software Design

Hardware-dependent Software for Multi- and Many-Core Embedded Systems

Rainer Dömer
University of California at Irvine
Irvine, CA 92697, USA
doemer@uci.edu

Andreas Gerstlauer
University of Texas at Austin
Austin, TX 78712, USA
gerstl@ece.utexas.edu

Wolfgang Müller
University of Paderborn
33102 Paderborn, Germany
wolfgang@acm.org

Abstract— Due to the rapidly increasing software content in embedded systems, Hardware-dependent Software (HdS) has become a critical topic in system design. In this paper, we provide a brief overview on the topic of HdS, discuss the issues and complexities involved in the design of HdS, and motivate the need for special attention to HdS in research and development.

I. Introduction

Microelectronics are ubiquitous and took over large parts of our daily life. Together with an increasing profileration of such embedded systems, their complexities are growing exponentially. Mobile phones advanced to highly equipped communication and computing devices with support for multiple protocols like GSM 850/900/1800/1900, UMTS, IEEE 802.11b/g, Bluetooth, USB, and IrDA. Similarly, modern cars are typically equipped with 30-70 electronic control units (ECUs) connected by different networks like LIN, CAN, or FlexRay. In all cases, at the same time that system complexities in general are growing, the significance of embedded software is increasing at an even higher rate. During modern system development, up to 80% of the system content and functionality is implemented in software. Embedded software, that resides on the processors and microcontrollers, may already exceed 9GB in size [3]. These developments indicate that Hardware-dependent Software (HdS) has significantly gained relevance in embedded systems and Systems-on-Chip (SoCs) design, mainly due to its flexibility, the possibility of late change, and the quick adaptability.

The importance of HdS was early observed by the Virtual Socket Interface Alliance (VSIA) in 2002. The resulting outcome and taxonomy is summarized and further developed in [1]. However, until today only very few text books [4] provide complete overviews of the HdS area. The remainder of this paper aims to provide an introduction to HdS, the layer of software in embedded systems that directly interacts with the underlying hardware platform. We will introduce the key components of HdS and outline the various aspects in designing HdS for embedded systems.

A. Design Productivity Gap

The major driver behind electronic systems design in general, and HdS design in particular, is the design productivity gap that we are facing for a number of years now, and that, despite of many great efforts on all fronts to overcome the gap, still keeps growing.

The hardware design community is well-aware of the productivity gap in hardware design. For many years now, the potential capacities in chip size outpace the capabilities of designing these chips.

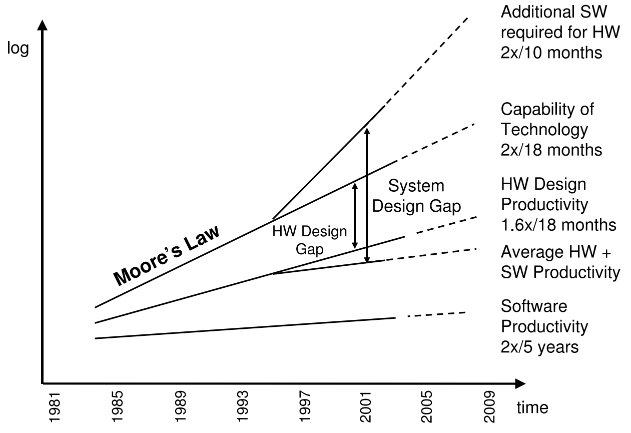

Fig. 1. Hardware- and software productivity gaps combined form the system design gap (source: Ecker et. al. [3]).

In particular, the capacity in chips doubles about every 18 months according to Moore's law [6], whereas the productivity growth in hardware design over the last years is estimated at only 1.6x over 18 months [10]. This gap in productivity growth, the so-called hardware design gap, is illustrated in the center of Figure 1 [3].

As embedded systems consist of both hardware and software, both productivity factors need to be taken into account when we aim to design entire systems. Due to Humphrey [5], we can identify a growth of software functions of 10x every 10 years [5]. Considering that additional software productivity gap, the situation for entire systems only gets worse. In this context, the actual needs in embedded software complexity would require an estimated growth of 2x over 10 months in order to satisfy the complexity involved in building real systems [3].

B. System Design Gap

Fact is, that we actually face two design gaps at the same time, a software design gap in addition to the well-known hardware design gap. Combining both productivity gaps, as shown in Figure 1, result in a large system design gap.

Moreover, additional complexity is created from the close interaction and tight dependency between the software and hardware domains. In other words, the necessary interfacing of software and hardware adds yet another layer of complexity. Thus, *Hardware-dependent Software (HdS)* is at the core of this system design challenge, as it deals exactly with those parts of the embedded software that interact directly with the underlying hardware.

II. Hardware-dependent Software

Hardware-dependent Software (HdS) can be defined as the software in an embedded system that closely interacts with the underlying hardware platform [3].

Typically, HdS is specifically built for a particular block of hardware, and both the HdS and the hardware together implement a systems' functionality. As such, HdS provides the application software with an interface to easily access those features which are directly supported by hardware components.

Looking at HdS from the perspective of software architecture, we can identify HdS as a layer of software modules in between the application software and the underlying hardware platform. In other words, HdS can very well be seen as low level software.

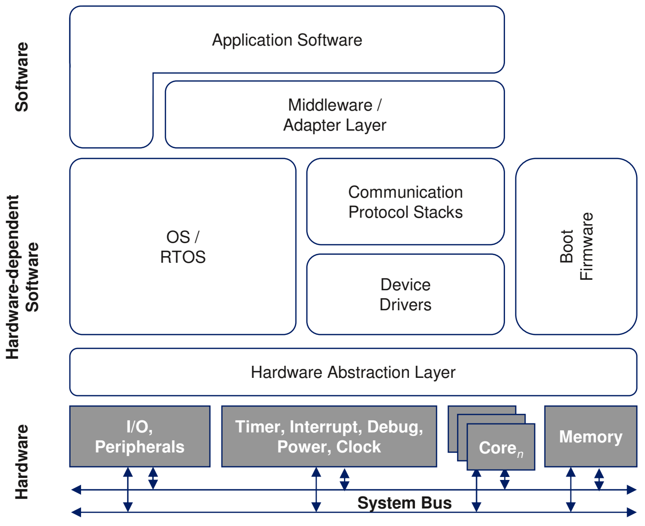
Software
Application Software
Middleware / Adapter Layer
Hardware-dependent Software
OS / RTOS
Communication Protocol Stacks
Device Drivers
Boot Firmware
Hardware Abstraction Layer
Hardware
I/O, Peripherals
Timer, Interrupt, Debug, Power, Clock
Core$_n$
Memory
System Bus

Fig. 2. HdS in a layered software architecture (based on Ecker et. al. [3]).

Figure 2 shows the layering of the general HdS architecture. It should be clear that the layered HdS architecture shown is conceptual and generic. In many cases, depending on the actual embedded application, the software layers present will vary widely.

At the top portion of the figure, the application software and middleware layers are supported by a layer of various HdS modules in the middle. The HdS layer, in turn, is supported by the underlying hardware at the very bottom of Figure 2.

HdS typically runs in the kernel space of an operating system, whereas middleware and application software run in user space. As such, HdS includes the software modules for boot code, device drivers, hardware-dependent portions of protocol stacks, and DSP algorithms.

As shown in Figure 2, the embedded software stack is conceptually composed of several main components: *Application software* implements the functionality of the system, consisting of multiple processes and/or threads. *Middleware* represents a software layer for application-specific services and acts as an adapter layer between the application software and the operating system. The *real-time operating system* (RTOS) manages the application tasks and allows the sharing of available resources. *Communication protocol stacks* operate on top of device drivers which in turn provide the access to hardware resources. *Boot firmware* support the initial boot process of the embedded computer system. Today, most desktops and servers are BIOS-based whereas embedded system typically come with proprietary boot firmware[1]. Finally, the *hardware abstraction layer* (HAL) is a software layer used in many embedded contexts that provides an abstract interface to access the actual hardware resources.

RTOS, communication protocol stacks (including device drivers and interrupt handlers), boot firmware and HAL together form the hardware-dependent software stack.

III. Developing HdS

HdS development tools and design flows are typically specific to individual application areas. Nevertheless, common approaches across different areas can be identified.

Today, in many cases developing HdS means software coding by classical dedicated C development environments and targeted tool chains, i.e., C cross-compilers, off/online debuggers, linters, and target-specific assemblers and linkers. Due to its proximity to the underlying hardware, HdS in many cases needs to be able to access the hardware directly. HdS development therefore requires target-specific extensions of the language and tool chain beyond standard ANSI C using mechanisms such as intrinsics, pragmas or inline assembly. Such proprietary extensions coupled with the fact that C inherently exposes many machine-specifics, makes the HdS development process highly specific, error-prone and tedious. While a HAL aims to mitigate such effects by allowing the majority of the HdS to be developed independent of the underlying hardware, there is an on-going need for development solutions at higher levels of abstraction.

For some domains, we can observe that Matlab and Simulink were accepted as a pseudo-standard in order increase productivity. With the application of Matlab, the notion of model-based design was introduced[2]. In this context, model-based design referred to Simulink and Stateflow models for modeling

[1]We currently observe a migration from BIOS to EFI (Extensible Framework Interface). Apple Computers already moved to EFI with their Intel platforms, and MSI recently introduced its first EFI board.

[2]At the same time, the OMG introduced their concepts of Model-Driven Architecture which are different from those introduced by Matlab.

and as front-ends for advanced code generators which may be certified along RTCA DO-178B or IEC 61508. Examples for C code generation are the Real-Time Workshop by MathWorks and TargetLink by dSPACE. However, it should be noted that such model-based design and code generation concepts are typically reserved for higher layers of the software stack such as the application level. In constrast, due to the lack of control over the hardware, they may apply to HdS development only to a lesser degree.

Similar, automatic generation-based approaches can be applied to different aspects of HdS development using more domain-specific solutions. In all cases, the idea is to automatically generate low level HdS code from some abstract, high-level input description. For example, there are approaches for automatic assembly of customized RTOS implementations from an abstract selection of OS functionality required for a specific design instance [8]. Similarly, based on a high-level description of device registers, basic firmware to access hardware functionality can be automatically implemented [2]. Furthermore, complete protocol stacks and device drivers can be automatically synthesized to provide desired abstract communication primitives such as message-passing or semaphores in communication with external hardware [9]. In all cases, the goal of such on-going research is to provide solutions for automated HdS development across different target instances without the need for manual coding and associated debugging and validation costs.

IV. HdS for Multi- and Many-Core Platforms

The introduction of multi- and many core platform will certainly have considerable impact on embedded systems and their HdS. Studies from Venture Development Corp. (VDC), for instance, project a six time increase of the multicore microprocessors market between 2007 and 2011. Due to different application areas multi- and many-core platforms may cover symmetric, homogeneous architectures as well as combinations of heterogeneous cores with different parameters. In all cases, however, one or more cores are combined into a complete processor together with other hardware resources. At least a subset of these resources, such as interrupt controllers, L2 caches, or system bus interfaces, are typically shared among the cores. Depending on the number of cores, a processor with more than one core is typically called a multi-core (2-10 cores) or many-core (tens, hundreds, or thousands of cores) processor.

In each processor, a single operating system as part of a common, shared HdS stack manages threads or processes across cores[3]. Proper initialization, access and management of shared or replicated resources across cores and among processes/threads is one of the main complexity challenges in multi-/many-core firmware, driver, OS/RTOS and hence HdS development. For example, only recently have the first real-time scheduling implementations for multi-core processors emerged by RTOS vendors such as QNX [7]. As we are moving towards massively parallel many-core contexts, exponentially growing complexities in relation to race conditions, synchronization, and thread/process interactions will make the efficient implementation of necessary fairness and real-time guarantees a tremendous challenge.

[3]In contrast, when assembled into a multi-processor system, OS/RTOS and HdS stacks in each single-, multi- or many-core processor manage application thread/processes either standalone or in a distributed fashion across processors and eventually their cores in a hierarchical fashion.

V. Conclusion

Multi- and many-core HdS implementation, debugging, and verification impose significant complexities in the context of reliable and safe embedded real-time systems. Closing the productivity gap in system design is a challenge. The biggest obstacles may remain the efficient use of parallel computing resources and the verification of complete systems. However, this heavily relies on the specific application. Executing independent functions on different cores may help to reduce the number of microcontrollers. Challenges significantly increase with increased dependencies between those functions. Then, efficient inter-core and inter-chip communication is required which is not sufficiently supported by current tools. Current efforts on virtualization of resources provide promising directions to solve the main challenges.

References

[1] B. Bailey, G. Martin, and T. Anderson. *Taxonomies for the Development and Verification of Digital Systems*. Springer, 2005.

[2] W. Ecker, V. Esen, T. Steininger, and M. Velten. HW/SW interface - implementation and modeling. In W. Ecker, W. Müller, and R. Dömer, editors, *Hardware-dependent Software - Principles and Practice*. Springer, 2008.

[3] W. Ecker, W. Müller, and R. Dömer. Hardware-dependent software - introduction and overview. In W. Ecker, W. Müller, and R. Dömer, editors, *Hardware-dependent Software - Principles and Practice*. Springer, 2008.

[4] W. Ecker, W. Müller, and R. Dömer. *Hardware-dependent Software - Principles and Practice*. Springer, 2008.

[5] W. S. Humphrey. The future of software engineering: Part V. *Software Engineering Institute*, First Quarter 2002.

[6] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, April 1965.

[7] QNX Software Systems. QNX Neutrino RTOS. http://www.qnx.com/products/neutrino_rtos/, 2008.

[8] F. Rammig, M. Ditze, P. Janacik, T. Heimfarth, T. Kerstan, S. Oberthuer, and K. Stahl. Basic concepts of real time operating systems. In W. Ecker, W. Müller, and R. Dömer, editors, *Hardware-dependent Software - Principles and Practice*. Springer, 2008.

[9] G. Schirner, A. Gerstlauer, and R. Dömer. Automatic generation of hardware dependent software for MPSoCs from abstract system specifications. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, Seoul, Korea, Jan. 2008.

[10] Sematech Inc. International technology roadmap for semiconductors (ITRS), 2004 update, design. http://www.itrs.net, 2004.