LETTER

An RTOS-Based Design and Validation Methodology for Embedded Systems

Hiroyuki TOMIYAMA[†a)], ***Member***, **Shin-ichiro CHIKADA**[†], **Shinya HONDA**[†], ***Nonmembers***, ***and*** **Hiroaki TAKADA**[†], ***Member***

SUMMARY This paper presents an RTOS-based methodology for design and validation of embedded systems. The heart of our methodology is the use of an RTOS simulation model from the very early stage of the system design. A case study with a JPEG decoder application is also presented in order to demonstrate the effectiveness of our methodology.

key words: *RTOS, cosimulation, embedded systems*

1. Introduction

The functionalities of contemporary embedded systems have been continuously increasing. For example, recent cellular phones are much more than just telephones. They are now capable of sending/receiving emails, browsing world-wide-web pages, taking pictures, recording/playing movies, showing TV programs, displaying a map around user's current position using GPS, and so on. In such complex embedded systems, real-time operating systems (RTOSs) play an important role in managing the entire systems, i.e., both software and hardware. In this paper, we present an RTOS-oriented methodology for design and validation of embedded systems. The heart of our methodology is the use of an RTOS simulation model from the very early stage of the system design. Due to this, the concurrent and interactive behavior of the system can be specified accurately, validated efficiently, and implemented smoothly. The methodology is based on a flexible cosimulation platform and a complete simulation model of a standard RTOS which we have developed earlier [1]. In [1], we focused on the implementation of the cosimulator, but did not describe how to use it in the context of embedded system design. This paper, on the other hand, presents a practical methodology for design and validation of embedded systems.

This paper is organized as follows. Section 2 presents some related work. Section 3 describes our design and validation methodology, and Sect. 4 shows a case study. Section 5 concludes this paper with the current status and future work.

2. Related Work

In the past studies on hardware/software codesign and cosimulation, little attention has been paid to RTOSs regardless of their important role in embedded systems. Many earlier studies assume that codesign starts with system specification written in a software programming language such as C/C++. However, specifying the system functionalities in C/C++ alone is not feasible since C/C++ cannot capture the concurrent execution of multiple tasks which communicate with each other. In fact, these early research efforts assume a very simple system model consisting of a single or a few application task(s), and do not take account of RTOSs. Some other studies assume that the system specification is represented as abstract models such as task graphs. Using these models, it is possible to describe the concurrency without the help of RTOSs. However, most of the past studies using the abstract models did not provide any practical solution to simulate the specification. In the design of complex embedded systems, however, it is very important to extensively simulate the system specification in order to validate the functional correctness.

Several recent research efforts use generic RTOS models for system-level design and cosimulation [2]–[4]. The RTOS models are used for native execution of application software including RTOS service calls. After simulation-based validation, the RTOS calls are replaced with the system calls of the actual RTOS to be used in the final implementation to obtain the final software code. In [5], a method which automatically generates RTOS-dependent software from SystemC description is presented. The method replaces SystemC's constructs for concurrency and communication with corresponding RTOS service calls. In this sense, it can be considered that SystemC involves a simple RTOS model in itself. A common weakness of these RTOS models is that they support only a limited set of RTOS services in order to make the models generic and independent of specific RTOSs. For example, the RTOS model in [3] supports only 16 service calls. However, even small kernels of actual RTOSs provide much more services. For example, Standard Profile of μITRON [8], [9], i.e., one of the most popular RTOS kernels for small-scale embedded systems, has more than 80 service calls. These services may need to be fully utilized in order to write high-quality software. Therefore, it is easily imagined that the quality of software automatically generated by these previous methods is lower than that of handcrafted RTOS-dependent software.

In the design and validation methodology proposed in this paper, a designer first selects an RTOS and then de-

Manuscript received January 6, 2005.

[†]The authors are with the Department of Information Engineering, the Graduate School of Information Science, Nagoya University, Nagoya-shi, 464–8603 Japan.

a) E-mail: hiroyuki@acm.org

DOI: 10.1093/ietisy/e88–d.9.2205

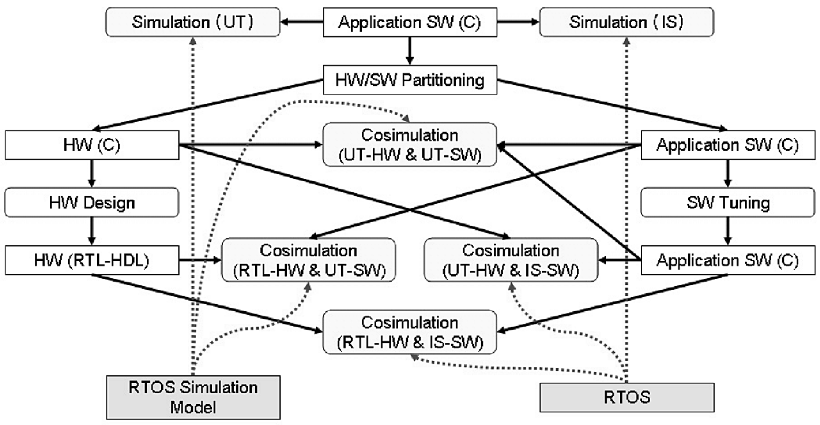

Fig. 1 The RTOS-based design and validation methodology.

scribes system specification using service calls provided by the RTOS. Since the specification depends on a specific RTOS, it is not easy to reuse the specification to different RTOS platforms. To our experience, however, most of design teams in industry do not use a variety of RTOSs in practice. They tend to keep using a very limited set of RTOSs for several years in order to reuse existing software. Our methodology is suited to such design teams or application domains where one or a very few RTOSs are used for a long period. Actually, we employ a standardized RTOS, i.e., μITRON, to develop CAD tools (the cosimulation platform and the RTOS simulation model, etc.) which support our methodology. μITRON is one of the most popular RTOSs in Japan for small- to middle-scale embedded systems. μITRON is not a specific RTOS product, but is an application programming interface (API) standard. It only defines a set of API functions, and implementation of the function bodies may differ among μITRON-based RTOSs. It is reported in [9] that over 40% of RTOSs used in Japan are based on the μITRON standard. Thus, our methodology is effective as long as a designer keeps using RTOSs which conform to the μITRON standard. Note that our methodology itself is not limited to μITRON-based RTOSs. Other RTOSs can be used if there exist a simulation model for the RTOSs.

In [6] and [7], a different approach to embedded software design is presented where an application-specific RTOS and its simulation model are automatically generated. In their approach, application software is analyzed first, and then only the RTOS services used in the application are included in the final RTOS. The work is similar to ours in that a set of services which can be used in application software is pre-defined. The major difference is that their work puts a special focus on customizing an RTOS while our methodology is based on a standard RTOS.

3. The RTOS-Based Design and Validation Methodology

Figure 1 illustrates our overall methodology for design and validation of embedded systems. In our methodology, system design starts with deciding an RTOS and describing system specification in C as a set of application tasks. Concurrency and inter-task communication are described by means of service calls of the RTOS. Thus, the system specification is in actual a software implementation of the system functionality. In order to validate the correctness of the application software, the software is compiled and linked with the RTOS simulation model to generate object code, which is then executed on the host computer. Due to the native execution, the simulation speed is very high. If a design error is found, the software is modified, and then the simulation is executed again. In Fig. 1, this simulation is denoted as "Simulation (UT)" where UT stands for the untimed level.

Next step is to evaluate the performance of the software. An instruction-set simulator (ISS) of the target processor is used in order to count the execution cycles. This process is denoted as "Simulation (IS)" in Fig. 1 where IS stands for the instruction-set level. If the performance does not meet the required level, a part of the system functionality needs to be implemented in hardware. The ISS is used to find the performance-critical portion to be moved from software to hardware. In Fig. 1, this process is denoted as "HW/SW Partitioning". At this level, cosimulation is performed in order to validate the functional correctness of the partitioning, where both the hardware and the software are directly executed on the host computer, i.e., denoted as "Cosimulation (UT-HW & UT-SW)" in Fig. 1.

After the cosimulation, register-transfer level (RTL) design of the hardware part and optimization of the software are started at the same time. The RTL description of the hardware may be obtained by either behavioral synthesis or manual design. The RTL design needs to be simulated to validate the functional correctness. At this stage, the RTL design is executed on an HDL simulator, while the software (i.e., application software and RTOS) is natively executed on the host computer. The software serves as a testbench which generates input data to and receives output data from the hardware in an interactive manner. This cosimulation is denoted as "Cosimulation (RTL-HW & UT-SW)" in Fig. 1.

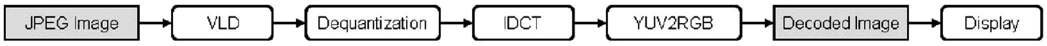

Fig. 2 A JPEG decoder example.

Concurrently with RTL design of the hardware, the application software is optimized to fully utilize the advantage of the hardware architecture. At this stage, the RTL design may not be completed. Therefore, the untimed hardware model is used for validation of the optimized software. The software is executed either natively or on the ISS. The latter case is denoted as "Cosimulation (UT-HW & IS-SW)" in Fig. 1.

After both the RTL hardware design and the software optimization are completed, cycle-accurate cosimulation is performed in order to validate the functional and temporal correctness of the system. This cosimulation is denoted as "Cosimulation (RTL-HW & IS-SW)" in Fig. 1.

The heart of our methodology is the use of an RTOS simulation model from the very early stage of the system design. Due to this, the concurrent and interactive behavior of the system can be specified accurately, validated efficiently, and implemented smoothly. Another key issue is the flexible cosimulation platform which supports various abstraction levels for both hardware and software.

In order to realize the proposed methodology, we have developed a cosimulation platform and a simulation model of an RTOS [1]. The RTOS simulation model supports all of the service calls which are defined by μITRON 4.0 Standard Profile. The RTOS model is implemented in C, so that it is directly executable on the host computer. The cosimulation platform is very flexible. It features plug-and-play of an ISS, HDL simulators, and functional hardware models in C/C++. At present, our cosimulation environment supports μITRON-based RTOSs only. However, other RTOSs can be easily supported if there exist simulation models for the RTOSs.

4. A Design Example

In order to evaluate the effectiveness of our RTOS-based design and validation methodology, we designed a JPEG decoder system. Figure 2 shows the flow of the JPEG decoder where there are four tasks: VLD, Dequantization, IDCT, and YUV2RGB. In addition, there is another task, Display, to display the decoded image on a window of a host computer during simulation. The simulation was performed on dual Xeon 2.4 GHz processors with hyper-threading technology, running on MS-Windows XP. The cosimulation platform and the RTOS simulation model presented in [1] were used in this case study.

First, we designed all the tasks in the C language. The tasks were compiled and linked together with the RTOS simulation model to generate binary code which is executable on the host computer. By executing the binary code, we validated the functional correctness of the JPEG program. The simulation time (CPU time) was 9 milliseconds.

Next, we ran simulation on an instruction-set simulator (ISS) in order to estimate the execution time of the JPEG application on the target processor. We used the ARM9 core [11] as a target processor. We compiled the JPEG program using a cross-compiler from ARM Corporation, and linked it with an actual RTOS, i.e., the TOPPERS/JSP kernel [12]. Then, the generated code was executed on an ISS, i.e., ARMulator [11]. The simulation took 23.8 seconds. It should be stressed that we did not have to change the application program at all because of our complete RTOS simulation model. The simulation result showed that the execution time of the JPEG application on the target processor would be 62.7 million cycles.

From the result of simulation using the ISS, we found that 39.8% of total execution time was spent by the IDCT task. In order to enhance the performance, we decided to implement the IDCT task in hardware. We created a functional hardware model of IDCT in the C language by copying the body of the original IDCT task into the hardware model. Additional code for hardware/software communication was inserted at the entry and exit points of the IDCT model. In the software task for IDCT, the body was replaced with RTOS service calls for memory-mapped I/O access and synchronization between the software and the hardware. Then, cosimulation was performed. The software tasks were compiled and linked with the RTOS simulation model to generate an object file. The IDCT hardware model was also compiled to obtain an object file. Then, the two files were executed on our cosimulation platform in order to validate the functional correctness of the system. The simulation time was 46 seconds on the host computer.

Having validated the functional correctness, we started RTL design of the IDCT circuit. We designed the IDCT circuit in VHDL. In order to validate the functional correctness of the IDCT design, we ran cosimulation. At this phase, the IDCT design was executed on an HDL simulator, i.e., ModelSim [10], while the software was run directly on the host computer. This cosimulation took 182 seconds. Again, it should be stressed that we did not have to change the application software.

Concurrently with designing the IDCT circuit, we can optimize the application software and run cosimulation to verify the correctness of the optimized software on the ISS. In this experiment, however, we did not optimize the software. We just ran cosimulation to validate the behavior of the application software on the ISS. Since RTL design of the IDCT circuit had not been completed yet at that time, the functional C model was used for the cosimulation. The cosimulation time was 71 seconds.

Having completed the IDCT design in VHDL, we finally performed cosimulation using the HDL simulator and the ISS. From the cosimulation, we found that the execution cycles of the JPEG application was 39.1 million cycles. This means that the execution time was shortened by 38% compared with the software-only implementation. This cosimulation required 248 seconds.

As we have seen, cosimulation time was largely different depending on the abstraction level of both software and hardware. In our design and validation methodology, cosimulation can be performed at just the necessary level of abstraction.

5. Conclusions

In this paper, we have presented an RTOS-based design and validation methodology for embedded systems. The heart of our methodology is the use of an RTOS simulation model from the very early stage of the system design, which enables accurate system specification, efficient validation, and smooth implementation. We also showed a case study in order to demonstrate the effectiveness of our methodology.

Up to now, we have developed a cosimulation framework and a simulation model of RTOS [1] which are key components to realize the proposed methodology. Currently we are extending our RTOS model and cosimulation platform to support multiprocessor embedded systems.

Acknowledgment

This work was partially supported by JSPS Grant-in-Aid for Scientific Research (B) #17300014.

References

[1] S. Honda, T. Wakabayashi, H. Tomiyama, and H. Takada, "RTOS-centric cosimulator for embedded system design," IEICE Trans. Fundamentals, vol.E87-A, no.12, pp.3030–3035, Dec. 2004.

[2] D. Desmet, D. Verkest, and H. De Man, "Operating system based software generation for systems-on-chip," Proc. Design Automation Conference (DAC), pp.396–401, 2000.

[3] A. Gerstlauer, H. Yu, and D. Gajski, "RTOS modeling for system level design," Proc. Design Automation and Test in Europe (DATE), Embedded Software Forum, pp.130–135, 2003.

[4] H. Yu, A. Gerstlauer, and D. Gajski, "RTOS scheduling in transaction level models," Proc. Int'l Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), pp.31–36, 2003.

[5] F. Herrera, H. Posadas, P. Sanchez, and E. Villar, "Systematic embedded software generation from SystemC," Proc. Design Automation and Test in Europe (DATE), Embedded Software Forum, pp.142–147, 2003.

[6] L. Gauthier, S. Yoo, and A.A. Jerraya, "Automatic generation and targeting of application-specific operating systems and embedded systems software," IEEE Trans. Comput. Aided Des. Integr. Circuits Syst., vol.20, no.11, pp.1293–1301, Nov. 2001.

[7] S. Yoo, G. Nicolescu, L. Gauthier, and A.A. Jerraya, "Automatic generation of fast timed simulation models for operating systems in SoC design," Proc. Design Automation and Test in Europe (DATE), pp.620–627, 2002.

[8] H. Takada and K. Sakamura, "μITRON for small-scale embedded systems," IEEE Micro, vol.15, no.6, pp.46–54, 1995.

[9] ITRON, http://www.assoc.tron.org/itron/

[10] Mentor Graphics Corporation, http://www.mentor.com/

[11] ARM Corporation, http://www.arm.com/

[12] TOPPERS Project, http://www.toppers.jp/