Framework for Automated Application-Specific Optimization of Embedded Real-Time Operating Systems

Sindhwani, M and Srikanthan, T
Centre for High Performance Embedded Systems,
Nanyang Technological University, Singapore.
astsrikan@ntu.edu.sg

Abstract— **In recent years, embedded systems have become increasingly more complex. This complexity is tackled in software by abstracting the underlying hardware using an embedded real-time operating system (RTOS) and a suitable Board Support Package (BSP). However, the RTOS imposes overheads on the CPU in return for the run-time support it provides. Modern embedded hardware often comprises multi-core processors, unified core processors, soft-core processors, dedicated hardware logic, or even a system-on-chip. Each of these hardware options can be used to reduce the CPU overhead of the RTOS and numerous methods have been proposed in literature. Due to the large number of optimization options available and the need to meet strict time-to-market pressures, RTOS optimization needs to be a largely autonomous process. In this paper, we present a framework for automated application-specific optimization of embedded real-time operating systems. We identify the components of such a framework and discuss our prototype of the framework.**

Keywords—**real-time operating system, RTOS, optimization, framework, embedded systems**

I. Introduction

Recent technological advances have resulted in greater levels of computing power being made available for embedded systems design. At the same time, higher levels of integration and better tool support have made system-on-chip a feasible reality for large volume applications. The post-PC computing era has been enabled by cheaper computing power, cheaper memory, improved power profile of electronic components, greater levels of integration, and explosive growth in communications and network technologies.

In software, complexity has been managed by abstracting hardware using an RTOS and a suitable BSP. However, the RTOS imposes overheads on the CPU in return for the run-time support it offers [1]. Although methods to reduce CPU overheads of the RTOS have been proposed in literature, they typically require the system designer to treat the RTOS as yet another problem in the system design process. For a methodology to be applicable and feasible, it is imperative that it integrates seamlessly with the system design process. Also, RTOS optimization should be a largely autonomous process.

In this paper, we present a framework for automated application-specific optimization of embedded real-time operating systems, based on the instrumentation and simulation of RT-UML models of the application. We identify the components of such a framework and discuss our prototype of the framework.

Section 2 of this paper looks at the background and motivation for this framework. The current trends in hardware and software for embedded systems are briefly examined to set the context for the proposition. The need for such a framework is identified in Section 3. The actual framework is proposed in Section 4. Section 5 discusses our prototypes of the framework and presents our analysis. Finally, our conclusion is presented in Section 6.

II. Background and Motivation

In this section, we briefly look at the trends in hardware, software and system design for embedded systems. We also look at the trends in RTOS optimization and identify the problems facing developers of embedded systems.

A. Trends in Embedded Hardware

Embedded hardware has become increasingly more complex. Improvements in process and processor technology have resulted in significantly higher levels of integration in single chips, resulting in the following technology trends:

1. Multi-core Processors: Multi-core processors comprise more than one processing core on the same chip. The second processor may be an I/O co-processor [2], a signal processing core [3] or another processor core [4].
2. System on chip: High levels of integration have resulted in full systems being fabricated on the same chip. This includes the main CPU, memory, supporting digital logic, and analog and digital peripherals, and is a viable option for highly integrated high-volume applications.
3. Configurable system on chip: Due to the availability of high-density field programmable gate arrays (FPGA), a recent trend is the use of soft-core CPUs in FPGA space. This has resulted

in complete systems being created on an FPGA fabric. Soft-core processors, such as the NIOS [5], can be extended with application-specific custom instructions.

At the same time, the embedded systems industry has been pushed towards the post-PC computing era due to the development of numerous wireless communication technologies, energy- and power-efficient hardware and improved battery technology.

B. *Trends in Embedded Software*

There are two main trends in software for embedded systems.

1. Higher Levels of Abstraction: Due to the increasing complexity of the underlying hardware and the need to meet strict time-to-market pressures, software for embedded systems has largely focused on moving towards higher levels of abstraction. This has meant the increased use of embedded RTOS and board support packages (including device drivers).
2. Increased Focus on Re-use: To increase developer productivity, there is a greater focus on re-use. This has meant increased use of object-oriented languages, such as C++. Also, system developers frequently use third-party middleware, such as protocol stacks, in the design. Such middleware is usually pre-verified, pre-tested and meets rapidly evolving standards. Further, applications are also being coded in JAVA [6] to remain independent of the hardware and RTOS.

C. *Trends in Embedded System Design*

Embedded software designers are turning towards software design methodologies to manage the complexity inherent in modern embedded systems projects. Also, as software design tools have matured, they have been targeted towards embedded systems projects.

UML [7] is frequently used for the design of software for enterprise and desktop systems. However, in recent years, there have been Real-Time UML (RT-UML) tools [8, 9] that claim to be able to generate production quality code from RT-UML models. In order to deliver high quality solutions on time, there will be an increased emphasis on modeling and simulation for complex systems. At the same time, Time-To-Market (TTM) pressures will be met by the code generation capability of RT-UML tools.

D. *RTOS Optimization*

For complex systems running at low to medium clock frequencies, CPU overhead imposed by the RTOS can be as high as 25% [10]. Such an overhead is unacceptable and in [11], we have outlined the numerous methods proposed in literature to use features of modern embedded computing platforms to contain this overhead. Most of these methods rely on supporting the RTOS with custom hardware or adding custom instructions to the instruction set of a soft-core processor, as we demonstrated in [10].

Although numerous methods can be used to reduce CPU overheads imposed by the RTOS, different methods require different resources and provide varying benefits depending on the requirements of the target. In fact, if system specifications change, a different method of RTOS optimization may be applicable. This requires system developers to treat RTOS optimization as yet another problem that requires trade-off analysis and affects the time-to-market.

III. Need for an RTOS Optimization Framework

As discussed, numerous methods can be used to minimize RTOS overheads and the actual solution that should be used depends on the system being designed. The selection is influenced by the needs and capabilities of the embedded system. For example, a hardware scheduler should be used only if scheduling overheads in the system are significant. But this can be done only if hardware space is available in the target. Finally, the actual scheduler that can be used and its performance depend on the amount of available hardware space. We propose that RTOS optimization should be treated as a hardware/ software partitioning problem since it is non-optimal to say that “one size fits all”. In fact, it is important to factor in RTOS utilization by the embedded application and scale the RTOS to be application specific. In this context, application specificity and scalability considers not only the features that are used, but also the frequency with which each feature is used.

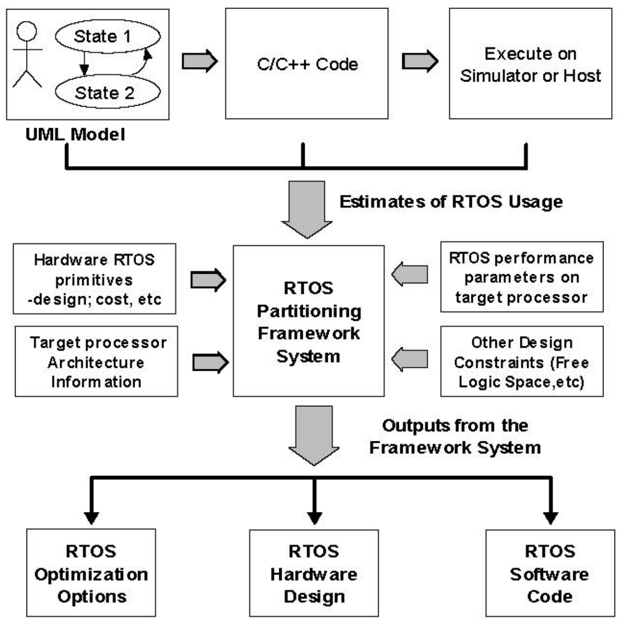
To these ends, we propose a framework within which it is possible to analyze and compare various software and hardware implementations of RTOS primitives. Such a framework could then be used to determine optimal or near-optimal solutions for RTOS optimization for the embedded system in question. Such a framework is proposed next.

IV. RTOS Optimization Framework

The proposed RTOS optimization framework is shown in Figure 1. As inputs, the system takes a set of requirements and a set of constraints. On the output side, it presents schemes for RTOS partitioning and also produces the hardware and software for the RTOS. A supporting database stores information about various software and hardware alternatives that can be used.

A. *Inputs to the Framework*

Since complex embedded systems are modeled in RT-UML, we propose using the RT-UML system specification as the primary input for our analysis. The code generated by the RT-UML code synthesis tool will be instrumented and executed on a host platform. These simulated runs will be used to extract both static and dynamic information about RTOS usage by the application. Static information will include information such as the number of tasks, semaphores and other RTOS resources used by the application. Information about modules that have not been modeled in RT-UML will be provided using an information file.



State 1
State 2
UML Model
C/C++ Code
Execute on Simulator or Host
Estimates of RTOS Usage
Hardware RTOS primitives -design; cost, etc
Target processor Architecture Information
RTOS Partitioning Framework System
RTOS performance parameters on target processor
Other Design Constraints (Free Logic Space,etc)
Outputs from the Framework System
RTOS Optimization Options
RTOS Hardware Design
RTOS Software Code

Figure 1. RTOS Optimization Framework

The RTOS framework driver will also be provided with a set of constraints to be used as parameters for the optimization of the system.

B. *A-priori Information*

A priori information, used by the RTOS partitioning framework for making a decision, will be collected and stored in a database. This will include:

- Target Processor Architecture Information: Information about the target hardware platform will include details such as the architecture of the target CPU, context switching overheads, reconfigurable options in the system, instruction set architecture (whether it can be modified), operating clock frequency, support for shared data protection, and available coprocessors. All this information will help the framework assess the cost associated with adding specific hardware for RTOS activities, and/ or splitting the RTOS into independent modules that run on sets of processors.
- Hardware RTOS Primitives: In the past, several attempts have been made to port either a part, or the whole of the RTOS to execute in hardware, mostly as a coprocessor. It is also expected that RTOS primitives will be created as Intellectual Property that can be licensed by designers. The RTOS partitioning framework pre-supposes the existence of such primitives for RTOS activities. The information passed to the framework will also include the cost of hardware primitives, such as code and data requirements, hardware area requirement, processing time issues and CPU requirements for supporting software. This information will be used by the system to determine the cost that the embedded system will incur if the hardware primitive is to be added into the design.
- RTOS Performance Parameters on Target Processor: The RTOS partitioning framework will also need information about the cost of executing the RTOS in software on the target processor. This will include typical RTOS performance parameters, such as interrupt latency, task switching time and typical overheads for executing the RTOS primitives in software. This information will be used to calculate the total cost of an all-software solution on the target processor.

C. *Main Processing Module*

The main processing module of the framework combines all the inputs and constraints received from the system designer and carries out intelligent processing to propose applicable RTOS optimization techniques. The following processes are carried out:

1. Combine all inputs and extract RTOS usage information. RTOS usage information will include static parameters (such as number of resources) and dynamic parameters (frequency of use of each resource).

2. Retrieve optimization options applicable to the target.

3. If applicable, scale optimization options based on number of resources (e.g. the number of tasks in the system affects the performance and hardware requirement for a hardware scheduler).

4. Restrict applicable options, based on target system constraints (e.g. based on available hardware space)

5. Perform optimization based on the available options.

6. Generate RTOS optimization report and other outputs for the system designer.

D. *System Outputs*

The various solutions are presented to the system designer with an estimation of the associated costs. The framework will output RTOS optimization options for the entire system. It will also produce the hardware description and the software code for the final proposed RTOS for the system.

V. PROTOTYPING THE FRAMEWORK

We have implemented a restricted prototype of the RTOS optimization framework. The current framework is based on MicroC/OS-II [12] running on the Altera NIOS soft-core processor [5]. RTOS optimization is performed using instruction-set customization. The system is modeled using I-Logix Rhapsody [8] that is capable of generating production quality code from the system models. In this section, we discuss some aspects of our implementation.

A. *Extracting RTOS Usage Information*

In our framework, RTOS usage information is extracted by using simulation of the generated application on a host platform. The deployment model for Rhapsody is shown below. The code generated from the model of the application is tied to an Object Execution Framework (OXF) [13] that abstracts the target RTOS, allowing the system to be deployed to different platforms without changing the model. This also allows the generated code to be simulated on a host platform, and subsequently deployed to a different target.

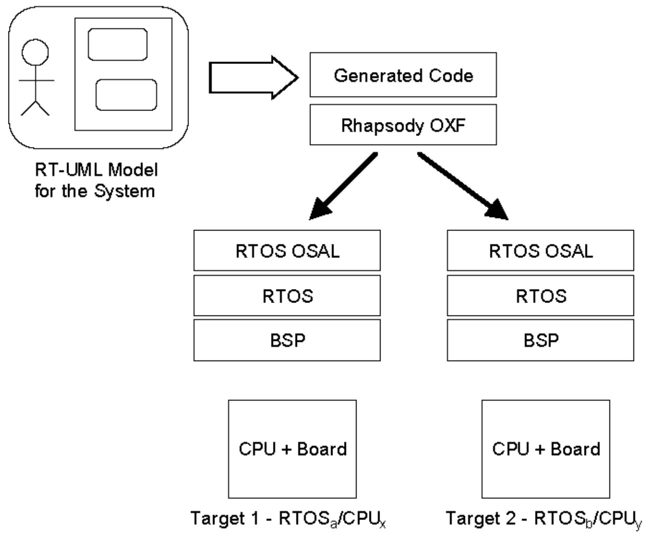


Figure 2. Rhapsody Deployment Model

In Rhapsody, an Operating System Adaptation Layer (OSAL) binds the abstract calls from the OXF to a specific RTOS. We instrumented the Rhapsody OSAL for Windows NT (our host platform) to extract RTOS usage information and export it as an XML file. The RTOS usage information collected is shown in Figure 3. By intelligent processing of the file, it is also possible to extract details such as task switches.

Task: Init, Start, Suspend, Resume,
Set priority, End
Timer: Init, destroy
Event Flag: Init, reset, signal, wait
Message Queue: Init, get, pend (wait), put
Mutex: Init, free, lock
Semaphore: Init, signal, wait

Figure 3. RTOS Usage Parameters Extracted

B. *RTOS Optimization Techniques*

For our prototype, we restricted ourselves to RTOS optimization based on our work with custom instructions for MicroC/OS-II on the NIOS platform [10].

C. *OSAL for MicroC/OS-II on NIOS*

To allow a seamless translation from the RT-UML model to executable code on our target, an OSAL was created for MicroC/OS-II running on the NIOS platform. Certain differences between the expectations of the Rhapsody OXF and the services provided by the MicroC/OS-II were taken into consideration.

D. *XML for Information Interchange*

We selected the Extensible Mark-up Language (XML) for knowledge interchange in the system for the following reasons:

1. Extensible Format: XML is extensible and is extremely useful for representing semi-structured information. XML allows the specification and use of an arbitrary number of tags and attributes to represent the information.

2. Easy to parse: XML was designed with the aim to ensure it was simple to parse. It is estimated that a computer science graduate should take about two weeks to write a parser for an XML file [15].

3. Available Parsers: Numerous optimized libraries are available in the public domain for parsing XML in programming languages such as C, C++, and Java, and also scripting languages, such as Perl and Python.

4. Textual Format: Being a textual format, XML is easy to parse on all platforms.

5. Human Readable: XML stores information as sets of attribute-value pairs for every element. Since the file is stored as text, it can be read with a text editor and a human can easily understand the knowledge stored in the file.

The use of XML also allows system developers to process the same information for different purposes. For example, a developer can use the simulation trace and profile it with a different parser to extract an RTOS selection criteria (based on the RTOS usage information) or the developer can use a different intelligence engine to do the optimization.

E. *Specifying Un-modeled Modules*

Modules that have not been modeled in RT-UML also need to be considered as part of the RTOS optimization process. This will include drivers from the Board Support Package and Off-the-shelf third-party Middleware.

To represent these components, we recommend using a "specification file" for each component. Three levels of specification are defined:

- Level 0 (required): Information about the static usage of RTOS resources – number of tasks, semaphores, etc. required by the module.
- Level 1 (optional): Typical Usage Frequency of the RTOS resources – for example, which semaphores are used more frequently.
- Level 2 (optional): Estimated RTOS usage based on application parameters – for example, a greater number of context switches will take place for the same serial port if the baud rate is higher. This information will be produced as an output from a "Usage Estimator" script provided by the vendor of the component.

Level 0 information is required but can be found from the porting documents of most middleware and can be specified by the system designer. Level 1 and Level 2 information would usually by available only from the vendor.

F. *Analysis of the Framework*

Our framework prototype covers many aspects of an RTOS optimization framework. It lets us model a system in RT-UML using Rhapsody and extract RTOS usage information through simulation of the application. Further, modules that are not modeled in Rhapsody can be represented using our proposed specifications.

The framework allows system developers to focus on activities such as modeling and simulation of the system. Outputs from the simulation trace are combined with the specifications of un-modeled software assets and used by the central driver to automatically determine the most applicable RTOS optimization scheme. Since all aspects of the system communicate using specific machine-readable formats, the framework can easily integrate with existing methodologies. If the system specification changes, results from a fresh simulation will automatically produce a new RTOS optimization scheme, if applicable. In this manner, we can do application-specific optimization of the RTOS.

VI. Conclusion

In this paper, we have proposed a framework for the automated optimization of embedded real-time operating systems (RTOS). We have taken the current state-of-the-art and the needs of system developers into consideration before making our proposition.

Our framework is based on the trace obtained through the simulation of code generated from RT-UML tools. Although static and dynamic behavior of the application is extracted by using instrumentation and simulation of the RT-UML model of the system, we have shown how software assets that have not been modeled in RT-UML can also be accommodated. We have prototyped a part of this framework that can be used with I-Logix Rhapsody, MicroC/OS-II and the Altera NIOS soft-core processor.

We believe that our proposition builds the basis for the development of a full-scale RTOS optimization framework.

References

[1] Rhodes D L, Wolf W, "Overhead Effects in Real-Time Pre-emptive Schedules", Proceedings of the seventh International Workshop on Hardware/Software Co-design, 1999, pp193-197.

[2] Infineon Technologies. "TriCore Architecture Manual version 1.3.2". (c) 2000.

[3] Renesas Technology, “SH-3/SH-3E/SH3-DSP Programming Manual”, 2000.

[4] Virtex II Pro Architecture. www.xilinx.com

[5] Altera NIOS CPU Data Sheet, March 2003.

[6] Sun Microsystems, “Java 2 Platform, Micro Edition (J2ME)” See: http://java.sun.com/j2me/

[7] http://www.UML.org

[8] I-Logix Inc., Rhapsody Product Information, available online at http://www.ilogix.com/rhapsody/rhapsody.cfm

[9] Artisan, Artisan Real-Time Studio, available online at: http://www.artisansw.com/products/products.asp

[10] Z Jin, M Sindhwani and T Srikanthan, “RTOS Acceleration on Soft-core Processors Using Instruction Set Customization”, 2004 IEEE International Conference on Field Programmable Technology (FPT 2004), Australia. Brisbane, Australia, pp. 371 – 374, Dec 2004.

[11] M Sindhwani, Tim Oliver, Douglas L Maskell and T Srikanthan, “RTOS Acceleration Techniques - Review and Challenges”, Proceedings of the Sixth Real-Time Linux Workshop, Singapore, pp. 123-128, Nov 2004.

[12] Labrosse J J, "MicroC/OS-II: the real-time kernel", Lawrence, Kansas R& D Publication, 1999.

[13] I-Logix Inc. “Rhapsody RTOS Adapter Guide – Rhapsody Release 2.3”. Part No. 2103.

[14] W3C, “Extensible Mark-up Language”, available at: http://www.w3.org/XML/

[15] XML.com, “XML: A Technical Introduction to XML”, available online at: http://www.xml.com/pub/a/98/10/guide0.html