Mälardalen University Licentiate Thesis
No.145

TOWARDS A PREDICTABLE COMPONENT-BASED RUN-TIME SYSTEM

Rafia Inam

January 2012

MÄLARDALEN UNIVERSITY
SWEDEN

Department of Computer Science and Engineering
Mälardalen University
Västerås, Sweden

Copyright © Rafia Inam, 2012
ISSN 1651-9256
ISBN 978-91-7485-054-3
Printed by Arkitektkopia, Västerås, Sweden
Distribution: Mälardalen University Press

Abstract

In this thesis we propose a technique to preserve temporal properties of real-time components during their integration and reuse. We propose a new concept of runnable virtual node which is a coarse-grained real-time component that provides functional and temporal isolation with respect to its environment. A virtual node's interaction with the environment is bounded by both a functional and a temporal interface, and the validity of its internal temporal behaviour is preserved when integrated with other components or when reused in a new environment.

The first major contribution of this thesis is the implementation of a Hierarchical Scheduling Framework (HSF) on an open source real-time operating system (FreeRTOS) with the emphasis of doing minimal changes to the underlying FreeRTOS kernel and keeping its API intact to support the temporal isolation between a numbers of applications, on a single processor. Temporal isolation between the components during runtime prevents failure propagation between different components.

The second contribution of the thesis is with respect to the integration of components, where we first illustrate how the concept of the runnable virtual node can be integrated in several component technologies and, secondly, we perform a proof-of-concept case study for the ProCom component technology where we demonstrate the runnable virtual node's real-time properties for temporal isolations and reusability.

We have performed experimental evaluations on EVK1100 AVR based 32-bit micro-controller and have validated the system behaviour during heavy-load and over-load situations by visualizing execution traces in both hierarchical scheduling and virtual node contexts. The results for the case study demonstrate temporal error containment within a runnable virtual node as well as reuse of the node in a new environment without altering its temporal behaviour.

To my husband Inam

Acknowledgments

First of all, I am grateful to my supervisors Professor Mikael Sjödin and Dr. Jukka Mäki-Turja without whose guidance and assistance this study would not have been successful. I specially thank Prof. Mikael Sjödin for his advices, invaluable inputs, support and encouragement, and always finding time to help me.

Many thanks go to Prof. Philippas Tsigas for informing me about the PhD position and encouraging me to apply at MRTC for a position.

I have attended a number of courses during my studies. I would like to give many thanks to Hans Hansson, Ivica Crnkovic, Mikael Sjödin, Thomas Nolte, Emma Nehrenheim, Daniel Sundmark, and Lena Dafgård for guiding me during my studies.

I want to thank the faculty members; Hans Hansson, Ivica Crnkovic, Paul Pettersson, Damir Isovic, Thomas Nolte, Dag Nyström, Cristina Seceleanu, Jan Carlson, Sasikumar Punnekkat, Björn Lisper, and Andreas Ermedahl for giving me vision to become a better student.

I would also like to thank to the whole administrative staff, in particular Gunnar, Malin, Susanne and Carola for their help in practical issues.

My special thanks also to all graduate friends, especially Sara D., Farhang, Andreas G., Aida, Aneta, Séverine, Svetlana, Ana, Adnan, Andreas H., Moris, Hüseyin, Bob (Stefan), Luis (Yue), Hang, Mikael, Nima, Jagadish, Nikola, Federico, Saad, Mehrdad, Juraj, Luka, Leo, Josip, Barbara, Antonio, Abhilash, Lars, Batu, Mobyen, Shahina, Giacomo, Raluca, Eduard, and others for all the fun and memories.

I want to thank Moris, Farhang, Notle, Jan, Jiří, and Daniel Cederman - whom I have enjoyed working with. I supervised the three master students, Mohammad, Sara A., and Wu. I wish them best of luck.

Finally, I would like to extend my deepest gratitude to my family. Many thanks go to my parents for their support and unconditional love in my life. My deepest gratitude goes to my husband Inam for being always positive and supportive in all these rough and tough days and to my daughters Youmna and Urwa for bringing endless love and happiness to our lives.

This work has been supported by the Swedish Foundation for Strategic Research (SSF), via the research programme PROGRESS.

Rafia Inam
Västerås, January, 2012

List of Publications

Papers Included in the Licentiate Thesis[1]

Paper A *Virtual Node – To Achieve Temporal Isolation and Predictable Integration of Real-Time Components.* Rafia Inam, Jukka Mäki-Turja, Jan Carlson, Mikael Sjödin. In the Global Science and Technology Forum: International Journal on Computing (JoC), Vol.1, No.4, 2011.

Paper B *Support for Hierarchical Scheduling in FreeRTOS.* Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, Seyed Mohammad Hossein Ashjaei, Sara Afshar. In Proceedings of the 16^{th} IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 11), pages 1-10, IEEE Industrial Electronics Society, Toulouse, France, September, 2011.

Paper C *Hard Real-time Support for Hierarchical Scheduling in FreeRTOS.* Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, Moris Behnam. In Proceedings of the 7^{th} International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT' 11), Pages 51-60, Porto, Portugal, July, 2011.

Paper D *Run-Time Component Integration and Reuse in Cyber-Physical Systems.* Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, Jiří Kunčar. MRTC report ISSN 1404-3041 ISRN MDH-MRTC-256/2011-1-SE, Mälardalen University, December, 2011.

[1]The included articles have been reformatted to comply with the licentiate layout.

Additional Papers, not Included in the Licentiate Thesis

Conferences and Workshops

- *A* Algorithm for Graphics Processors*. Rafia Inam, Daniel Cederman, Philippas Tsigas. In 3rd Swedish Workshop on Multi-core Computing (MCC'10), Gothenburg, Sweden, 2010.
- *Using Temporal Isolation to Achieve Predictable Integration of Real-Time Components*. Rafia Inam, Jukka Mäki-Turja, Jan Carlson, Mikael Sjödin. In 22nd Euromicro Conference on Real-Time Systems (ECRTS' 10) WiP Session, Pages 17-20, Brussels, Belgium, July, 2010.
- *Towards Resource Sharing by Message Passing among Real-Time Components on Multi-cores*. Farhang Nemati, Rafia Inam, Thomas Nolte, Mikael Sjödin. In 16th IEEE International Conference on Emerging Technology and Factory Automation (ETFA'11), Work-in-Progress session, Toulouse, France, September, 2011

Technical reports

- *An Introduction to GPGPU Programming - CUDA Architecture*. Rafia Inam. Technical Report, Mälardalen Real-Time Research Centre, Mälar dalen University, December, 2010.
- *Different Approaches used in Software Product Families*. Rafia Inam. Technical Report, Mälardalen Real-Time Research Centre, Mälardalen University, 2010.
- *Hierarchical Scheduling Framework Implementation in FreeRTOS*. Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, Seyed Mohammad Hossein Ashjaei, Sara Afshar. Technical Report, Mälardalen Real-Time Research Centre, Mälardalen University, April, 2011.

Contents

I

Thesis

Chapter 1

Introduction

In embedded real-time electronic systems, a continuous increasing trend in size and complexity of embedded software has observed during the last decades. To battle this trend, modern software-development technologies are being adopted by the real-time industry. One such technology is Component-Based Software Engineering (CBSE), where the system is divided into a set of interconnected components [1]. Components have well-defined functional interfaces which define both provided and required services. However, the functional interfaces do not capture timing behavior or temporal requirements. Further, the advent of low cost and high performance 8, 16, and 32-bit micro-controllers, have made possible to integrate more than one complex real-time components on a single hardware node. For systems with real-time requirements, this integration poses new challenges.

The aim of this thesis is to investigate techniques for predictable integration of software components with real-time requirements. Further the real-time properties of the components should be maintained for reuse in real-time embedded systems.

1.1 Research Problem

Temporal behavior of real-time software components poses difficulties in their integration. When multiple components are deployed on the same hardware node, the emerging timing behavior is unpredictable. This means that a component that is found correct during unit test may fail, due to a change in temporal behavior, when integrated in a system. Even if a new component is still operating correctly in the system, the integration could cause a previously integrated (and correctly operating) component to fail. Similarly, the temporal behavior of a component is altered if the component is reused in a new system. Since also this alteration is unpredictable, a previously correct component may fail when reused.

Further the reuse of a component is restricted because it is very difficult to know beforehand if the component will pass a schedulability test in a new system. For real-time embedded control systems, methodologies and techniques are required to provide temporal isolation so that the run-time timing properties could be guaranteed.

1.2 Proposal

In this thesis we address the challenges of encapsulating real-time properties within the components, in order to make the integration of real-time components predictable, and to ease component reuse in new systems. The purpose is to preserve the timing properties within the components thus component integration and reuse can be made predictable.

To achieve this, the real-time properties are encapsulated into reusable components, and hierarchical scheduling is used to provide temporal isolation and predictable integration among the components that further leads to the increased reusability of the components [2, 3, 4].

1.2.1 Runnable Virtual Node

We propose the concept of a *runnable virtual node*, which is an execution-platform concept that preserves the temporal properties of software executed in it [3]. It introduces an intermediate level between the functional entities (e.g. components or tasks) and the physical nodes. Thereby it leads to a *two-level deployment process* instead of a single big-stepped deployment; i.e. first deploying functional entities to the virtual nodes and then deploying virtual nodes to the physical nodes.

The virtual node is intended for coarse-grained components for single node deployment and with potential internal multitasking. We envision a handful of components (less than 50) per physical node. Hierarchical scheduling technique is embedded within the runnable virtual node to encapsulate the timing requirements within the components. Using an Hierarchical Scheduling

Framework (HSF) a subsystem (runnable virtual node in our case) are developed and analyzed in isolation, with its own local scheduler at first step of deployment and its temporal properties are validated. Then at the second step of deployment, multiple subsystems are integrated onto a physical node using a global scheduler without violating the temporal properties of the individual subsystems.

The runnable virtual node takes the advantages of both component-based software engineering and hierarchical scheduling approaches. It exploits encapsulation and reusability benefits of CBSE [1], and the temporal isolation and concurrent development and analysis of subsystems in isolation of HSF [5]. Moreover, combining the two approaches, results in the additional benefits of predictable integration and reuse of timing properties of the real-time components.

1.3 Contributions

The contributions presented in this thesis can be divided into two main parts:

HSF Implementation

HSF has attained a substantial importance since introduced in 1990 by Deng and Liu [6]. Numerous studies has been performed for the schedulability analysis of HSFs [7, 8] and processor models [9, 10, 11, 12] for independent subsystems. The main focus of this research has been on the schedulability analysis and not much work has been done to implement these theories.

We present our work towards an implementation of the hierarchical scheduling framework in an open source real-time operating system, FreeRTOS [13], to support temporal isolation among realtime components. We implement idling periodic and deferrable servers using fixed-priority preemptive scheduling at both local and global scheduling levels. We focus on being consistent with the underlying operating system and doing minimal changes to get better utilization of the system and keeping its API intact.

Allowing tasks from different subsystems to share logical resources imposes more complexity for the scheduling of subsystems. A proper synchronization protocol should be used to prevent unpredictable timing behavior of the real-time system. We extend the implementation of two-level hierarchical scheduling framework for FreeRTOS with the provision of resource sharing at two levels: (i) local resource sharing (among the tasks of the same subsystem) using the Stack Resource Policy (SRP) [14], and (ii) global resource sharing using the Hierarchical Stack Resource Policy (HSRP) [15] with three different methods to handle *overrun* (with payback, without payback, and enhanced overrun) [16]. Moreover, we extend the HSF implementation to use in hard-real time applications, with the possibility to include legacy applications and components not explicitly developed for hard real-time or the HSF.

We test our implementation on EVK1100 AVR32UC3A0512 micro-controller [17]. To test the efficiency of the implementation, we measure the overheads imposed by the HSF implementation during heavy-load and over-load situations. Moreover, we evaluate the overheads and behavior for different alternative implementations of HSRP with overrun from experiments on the board. In addition, real-time scheduling analysis with models of the overheads of our implementation is presented.

Presentation and Realization of Runnable Virtual Node Concept

Runnable virtual node is proposed as a means to achieve predictable integration and reuse of software components. Runnable virtual node is a coarse-grained real-time component encapsulating the timing properties and with potential internal multitasking. We present to utilize the hierarchical scheduling within the component-based technology to retain temporal properties, increasing predictability during components integration that further leads to the increased reuse of the real-time components. We believe that our idea can be easily generalized. We present how it can be applied to other commercial component-based technologies like AUTOSAR and AADL.

As a specific example, we realize the idea of runnable virtual node using the ProCom component technology and validate that its internal temporal behavior is preserved when integrated with other components or when reused in a new environment. Our realization of runnable virtual node exploits the latest techniques for hierarchical scheduling to achieve temporal isolation, and the principles from component-based software-engineering to achieve functional isolation. It uses a two-level deployment process (instead of a single big-stepped deployment) i.e. deploying functional entities to the virtual nodes and then deploying virtual nodes to the physical nodes, thereby preserving the timing properties within the components in addition to their functional properties. We perform a proof-of-concept case study, implemented in the ProCom component-technology executing on top of FreeRTOS based hierarchical scheduling framework to validate the temporal isolation among components and to test the reuse of components.

1.4 Background

This section presents the background technologies our work uses. We provide an overview of the ProCom component technology, used to realize the runnable virtual node concept. It is followed by an introduction of the hierarchical scheduling framework.

1.4.1 ProCom Component Model

Component-Based Software Engineering (CBSE) and Model-Based Engineering (MBE) are two emerging approaches to develop embedded control systems like software used in trains, airplanes, cars, industrial robots, etc. The ProCom component technology combines both CBSE and MBE techniques for the development of the system parts, hence also exploits the advantages of both. It takes advantages of encapsulation, reusability, and reduced testing from CBSE. From MBE, it makes use of automated code generation and performing analysis at an earlier stage of development. In addition, ProCom achieves additional benefits of combining both approaches (like flexible reuse, support for mixed maturity, reuse and efficiency tradeoff) [4].

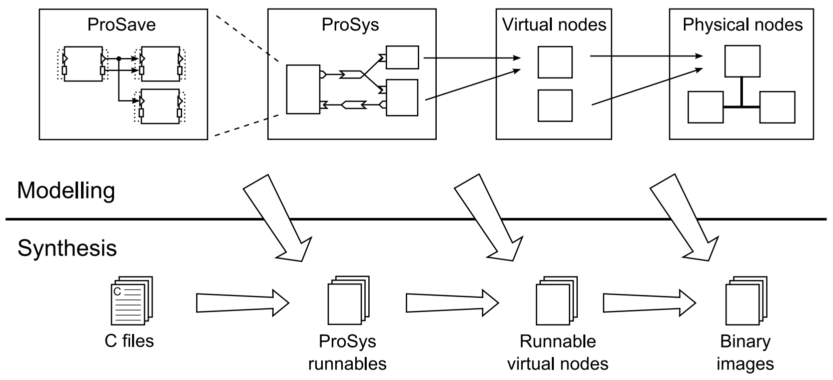


Figure 1.1: An overview of the deployment modelling formalisms and synthesis artefacts.

The ProCom component model can be described in two distinct realms: the modeling and the runnable realms as shown in Figure 1.1. In Modeling realm, the models are made using CBSE and MBE while in runnable realm, the synthesis of runnable entities is done from the model entities. Both realms are explained as follows:

The Modeling Realm

Modeling in ProCom is done by four discrete but related formalisms as shown in Figure 1.1. The first two formalisms relate to the system functionality modeling while the later two represent the deployment modeling of the system. Functionality of the system is modeled by the ProSave and ProSys components at different levels of granularity. The basic functionality (data and control) of a simple component is captured in ProSave component level, which is passive in nature. At the second formalism level, many ProSave components are mapped to make a complete subsystem called ProSys that is active in nature. Both ProSave and ProSys allow composite components. For details on ProSave and ProSys, including the motivation for separating the two, see [18, 19].

The deployment modeling is used to capture the deployment related design decisions and then mapping the system to run on the physical platform. Many ProSys components can be mapped together on a virtual node (many-to-one mapping) together with a resource budget required by those components. After that many virtual nodes could be mapped on a physical node i.e. an ECU (Electronic Control Unit). The relationship is again many-to-one. Details about the deployment modeling are provided in [4].

The Runnable Realm

At the runnable realm, runnables/executables are synthesized from the ProCom model entities. The primitive ProSave components are represented as a simple C language source code in runnable form. From this C code, the ProSys runnables are generated which contain the collection of operating system tasks. Virtual nodes, called runnable virtual nodes here, implement the local scheduler and contain the tasks in a server. Hence a runnable virtual node actually encapsulates the set of tasks, resource allocations, and a real-time scheduler within a server in a two-level hierarchical scheduling framework. Final binary image is generated by connecting different virtual nodes together with a global scheduler and using the middleware to provide intra-communications among the virtual node executables.

Deployment and Synthesis Activities

Rather than deploying a whole system in one big step, the deployment of the ProCom components on the physical platform is done in the following two steps:

- First the ProSys subsystems are deployed on an intermediate node called virtual node. The allocation of ProSys subsystems to the virtual nodes is many-to-one relationship. The additional information that is added at this step is the resource budgets (CPU allocation).
- The virtual nodes are then deployed on the physical nodes. The relationship is again many-to-one, which means that more than one virtual node can be deployed to one physical node.

This two-steps deployment process allows not only the detailed analysis in isolation from the other components to be deployed on the same physical node, but once checked for correctness, it also preserves its temporal properties for further reuse of this virtual node as an independent component. Chapter 3 describes this further.

The PROGRESS Integrated Development Environment (PRIDE) tool [20] supports the automatic synthesis of the components at different levels [21]. At the ProSave level, the XML descriptions of the components is the input and the C files are generated containing the basic functionality. At the second level, ProSys components are assigned to the tasks to generate ProSys runnables. Since the tasks at this level are independent of the execution platform, therefore, the only attribute assigned at this stage is the period for each task; which they get from the clock frequency that is triggering the specific component. Other task attributes like priority are dependent on the underlying platform, hence assigned during later stages of the synthesis. A clock defines the periodic triggering of components with a specified frequency. Components are allocated to a task when (i) the components are triggered by the same event, (ii) when the components have precedence relation among them to be preserved.

1.4.2 Hierarchical Scheduling Framework

Hierarchical scheduling has shown to be a useful approach in supporting modularity of real-time software [22] by providing temporal partitioning among applications. A two-level hierarchical scheduling framework [23] is used to provide the temporal isolation among a set of subsystems. In hierarchical scheduling, the CPU time is partitioned among many subsystems (or servers), that are scheduled by a global (system-level) scheduler. Each subsystem contains its own internal set of tasks that are scheduled by a local (subsystem-level) scheduler.

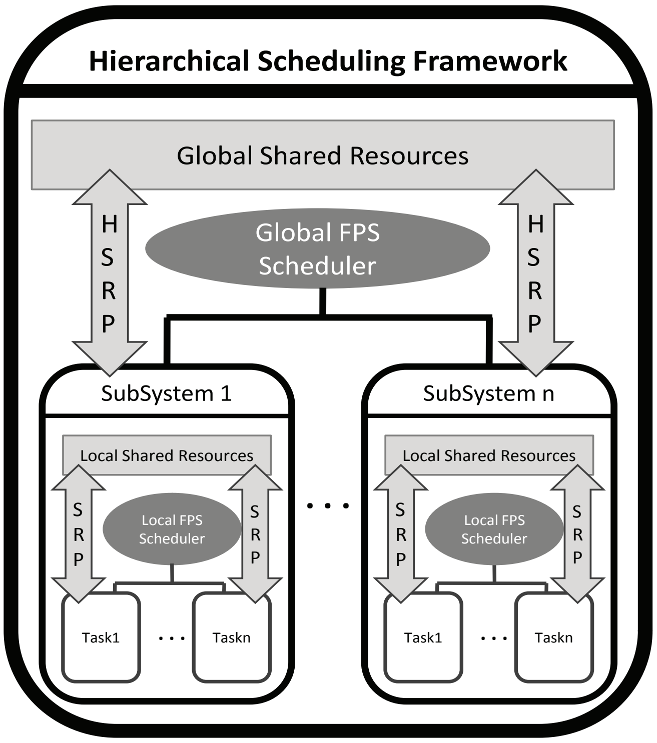


Figure 1.2: Two-level Hierarchical Scheduling Framework

Hence a two-level HSF can be viewed as a tree with one parent node (global scheduler) and many leaf nodes (local schedulers) as illustrated in Figure 1.2. The parent node is a global scheduler that schedules subsystems. Each subsystem has its own local scheduler, that schedules the tasks within the subsystem. The subsystem integration involves a system-level schedulability test, verifying that all timing requirements are met.

The HSF gives the potential to develop and analyze subsystems in isolation from each other [24]. As each subsystem has its own local scheduler, after satisfying the temporal constraints, the temporal properties are saved within each subsystem. Later, a global scheduler is used to schedule all the subsystems together without violating the temporal constraints that are already analyzed and stored in the subsystems. Accordingly we can say that the HSF provides partitioning of the CPU between different servers. Thus, server-functionality can be isolated from each other for, e.g., fault containment, compositional verification, validation and certification, and unit testing.

1.5 Thesis Overview

The thesis is organized in two distinctive parts. Part-I gives a summary of the performed research. Chapter 1 describes the motivation and background of the research. Chapter 2 formulates the main research goal, describes the research method we used, and introduces research questions used as guideline to perform the research. Chapter 3 describes our approach of runnable virtual node, and some results of our research. Finally Chapter 4 concludes the thesis by summarizing the contributions and outlining the future work.

Part-II includes three peer-reviewed scientific papers and one technical report contributing to the research results. These papers are published and presented in international conference and workshop, or international journals and are presented in Chapters 5-7. The technical report is submitted for conference publishing and is presented in Chapter 8. A short description and contribution of these papers and the report is given as follows:

Paper A. "Virtual Node: To Achieve Temporal Isolation and Predictable Integration of Real-Time Components". Rafia Inam, Jukka Mäki-Turja, Jan Carlson, Mikael Sjödin. In the Global Science and Technology Forum: International Journal on Computing (JoC), Vol.1, No.4, 2011.

Short Summary: This paper presents an approach of two-level deployment process for component models used in the real-time embedded systems to achieve predictable integration of real-time components. Our main emphasis is on the new concept of virtual node with the use of a hierarchical scheduling technique. Virtual nodes are used as means to achieve predictable integration of software components with real-time requirements. The hierarchical scheduling framework is used to achieve temporal isolation between components (or sets of components). Our approach permits detailed analysis, e.g., with respect to timing, of virtual nodes and these analysis is also reusable with the reuse of virtual node. Hence virtual node preserves real-time properties across reuse and integration in different contexts.

We have presented the methods to realize the idea of virtual node concept within the ProCom, AUTOSAR, and AADL component models.

Contribution: I initiated this journal paper. I was involved in most parts of this paper. It has been a joint effort between me and all the authors.

Paper B. "Support for Hierarchical Scheduling in FreeRTOS". In Proceedings of the 16^{th} IEEE International Conference on Emerging Technologies and Factory Automation (ETFA' 11). Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, Syed Mohammed Hussein Ashjaei, Sara Afshar. IEEE Industrial Electronics Society, Toulouse, France, September, 2011. **Awarded scholarship by IEEE Industrial Electronic Society as best student paper**.

Short Summary: This paper presents the implementation of hierarchical scheduling framework on an open source real-time operating system FreeRTOS to support the temporal isolation of a number of real-time components (or applications) on a single processor. The goal is to achieve predictable integration and reusability of independently developed components or tasks. It presents the initial results of the HSF implementation by running it on an AVR 32-bit board EVK1100.

The paper addresses the fixed-priority preemptive scheduling at both global and local scheduling levels. It describes the detailed design of HSF with the emphasis of doing minimal changes to the underlying FreeRTOS kernel and keeping its API intact. Finally it provides (and compares) the results for the performance measures of periodic and deferrable servers with respect to the overhead of the implementation.

Contribution: I was the initiator and author to all parts in this paper. I have contributed in the design of HSF implementation and have designed all the test cases and have performed the experiments. I supervised the students Mohammed and Sara who were responsible of the implementation part.

Paper C. "Hard Real-time Support for Hierarchical Scheduling in FreeRTOS". Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, Moris Behnam. In Proceedings of the 7^{th} International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT' 11), Porto, Portugal, July, 2011.

Short Summary: This paper presents extensions to the previous implementation of two-level Hierarchical Scheduling Framework (HSF) for FreeRTOS. The results presented here allow the use of HSF for FreeRTOS in hard-real time applications, with the possibility to include legacy applications and components not explicitly developed for hard real-time or the HSF.

Specifically, we present the implementations of (i) global and local resource sharing using the Hierarchical Stack Resource Policy and Stack Resource Policy respectively, (ii) kernel support for the periodic task model, and (iii) mapping of original FreeRTOS API to the extended FreeRTOS HSF API. We also present evaluations of overheads and behavior for different alternative implementations of HSRP with overrun from experiments on the AVR 32-bit board EVK1100. In addition, real-time scheduling analysis with models of the overheads of our implementation is presented.

Contribution: I was the initiator and the main author to majority parts in this paper. I have contributed in the design of HSF implementation and have designed all the test cases and have performed the experiments. Moris included the implementation overheads to the schedulability analysis and wrote that section.

Paper D. "Run-Time Component Integration and Reuse in Cyber-Physical Systems". Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, Jiří Kunčar. MRTC report ISSN 1404-3041 ISRN MDH-MRTC-256/2011-1-SE, Mälardalen University, December, 2011. **In submission for conference publishing (ICCPS'12).**

Short Summary: This paper presents the concept of runnable virtual nodes as a means to achieve predictable integration and reuse of software components in cyber-physical systems. A runnable virtual node is a coarse-grained real-time component that provides functional and temporal isolation with respect to its environment. Its interaction with the environment is bounded both by a functional and a temporal interface, and the validity of its internal temporal behavior is preserved when integrated with other components or when reused in a new environment.

Our realization of runnable virtual nodes exploits the latest techniques for hierarchical scheduling to achieve temporal isolation, and the principles from component-based software-engineering to achieve functional isolation. In the paper we present a proof-of-concept case study, implemented in the ProCom component-technology executing on top of FreeRTOS based hierarchical scheduling framework.

Contribution: I was the initiator and author to all parts in this paper. I have contributed in the design of the case study. Jiří was responsible of implementing the case study in Progress IDE, called PRIDE, and I executed it on the target platform using AVR Studio and performed all the tests and experiments.

Chapter 2

Research Overview

2.1 Research Goal

The aim of the thesis is to support management of real-time properties in component-based systems by achieving predictable integrations and reusability of those components. To achieve this, new methods and tools for embedded real-time component based systems has to be introduced where timing properties of the components can be preserved, and real-time components integration can be made predictable. Thus the main goal that the thesis aims is:

> *To provide methods to maintain the real-time properties of run-time components during their integration with other components and their reuse in a new environment.*

The goal is broad and is sub-divided into smaller research problems to solve it, which collectively approach the main goal as explained in the following section.

2.2 Research Methodology and Research Guiding Questions

This research work is carried out following the methodology based on the steps proposed/described by Shaw [25]. The main research activities we followed are:

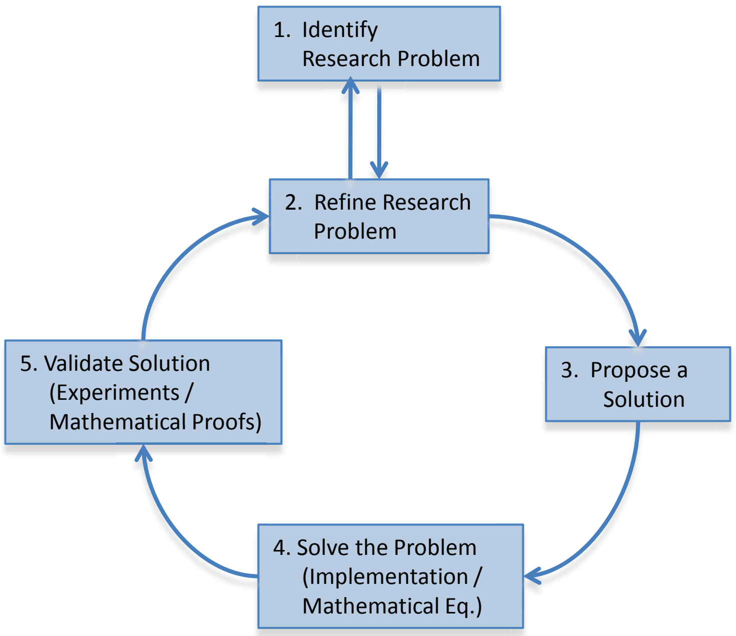

1. Identify Research Problem
2. Refine Research Problem
3. Propose a Solution
4. Solve the Problem (Implementation / Mathematical Eq.)
5. Validate Solution (Experiments / Mathematical Proofs)

Figure 2.1: The main research steps.

1. Study of the current state-of-the-art and identification of the research problem on current trends from real-time and component-based communities and definition of the research goal.

2. Refining the problem to research settings and defining research guiding questions.

3. Analysis of the current state-of-the-art in the literature based on guiding questions and proposing a solution.

4. Solving the proposed sub-problem (either by implementing a prototype and/or by providing mathematical analysis) and presenting the research results.

5. Illustration and validation of the research results. This is done by performing experimental evaluations of the implementation, and by per-

forming a case study. In case of mathematical analysis, the validation is done by implementation/formal proofs.

In this work, a big goal is identified at step 1, which is sub-divided into smaller research goals, and solved one at a time using steps 2 to 5. These steps (2 - 5) are performed repeatedly unless the desired results for the overall goal are achieved as described in Figure 2.1.

In the following section the research performed is described in brief.

- We have first identified and defined the initial problem.
- From this research, we performed some preliminary studies and formulated some initial research questions. Since our main focus is on maintaining the real-time properties within the components to attain their predictable integration, we first investigated within the real-time community for the suitable technique. Then we explored the possibilities to combine this technique in the component-based systems. It resulted in proposing a preliminary idea to merge the real-time technique into the component-based systems to achieve predictable integrations and reusability of those components [2, 3, 4]. The research questions arise here are Q1 and Q2, presented at the end of this section.
- Detailed study of state-of-the-art in relevant areas resulted in writing Paper A [2, 3] in which we proposed to use HSF technique within the virtual node component. HSF is known as a technique for providing temporal isolation between applications in real-time community. By embedding HSF within the component technology, temporal isolation and predictable integration of components can be achieved. As a result of this research, the runnable virtual node concept was introduced in the ProCom component model in the synthesis realm.
- Lacking support for HSF for our intended target-platform, we realized the need to implement HSF. This lead us to question Q3. We first implemented hierarchical scheduling framework in FreeRTOS real-time operating system independently from software components. We performed a detailed experimental evaluation on the implementation to test its temporal behavior and performance measures on the target platform an AVR-based 32-bit EVK1100 board. It resulted in writing Paper B [26].
- From this implementation, we figured out the need for hard real-time support in the FreeRTOS operating system. Also the resource sharing

protocols for HSF were needed to be implemented in order to provide resource sharing among components. Hence we extended our implementation with these properties and tested on the same target platform. We also showed to include the overheads of our implementation in the schedulability analysis of HSF. It resulted in writing Paper C [27].

- To realize the runnable virtual node, HSF implementation is integrated within the ProCom component model as a proof of concept. We now needed to validate our approach, as formulated by questions Q4 and Q5. We performed an example case study on the platform and run experiments to validate the temporal isolation among runnable virtual nodes, their smooth integration and reusability. The resulted paper, Paper D [28] is in submission.

While performing the above mentioned steps, we iteratively identified the following research questions which we used as a guideline for our research:

Q1 *Which techniques are used to achieve temporal isolations and predictable integration within the real-time community?*

Q2 *How can such a technique be integrated into component-technologies for embedded real-time systems?*

Q3 *How to efficiently implement hierarchical scheduling in a real-time operating system?*

Q4 *Can we demonstrate that we have achieved predictability and temporal isolations at run-time in real-time components and during their integration?*

Q5 *Can we demonstrate preservation of these temporal properties within the components when they are reused?*

Chapter 3

Runnable Virtual Node

In this chapter, we describe the runnable virtual node concept in brief and how it is used within the ProCom technology. We also provides some details of our HSF implementation that is used within the runnable virtual node. Finally, we provide some of the results obtained by executing an example case study using ProCom model as a proof-of-concept.

3.1 Runnable Virtual Node Concept

The concept of runnable virtual node is used to achieve not only temporal isolation and predictable temporal properties of real-time components but also to get better reusability of components with real-time properties. Components' reusability further reduces efforts related to software testing, validation and certification. This concept is based on a two-level deployment process as explained in Section 1.4.1. It indicates that the whole system is generated in two steps rather than a single big synthesis step. At the first level of deployment, the functional properties (functionality of components) are combined and preserved with their extra-functional properties (timing requirement) in the runnable virtual nodes. In this way it encapsulates the behavior with respect to timing and resource usage, and becomes a reusable component in addition to the design-time components. Followed by the second level of deployment, where these runnable virtual nodes are implemented on the physical platform along with a global scheduler.

A runnable virtual node includes the executable representation of the components assigned to the tasks, a resource allocation (period and budget of server), and a real-time scheduler, to be executed within a server in the hierarchical scheduling framework.

3.2 HSF Implementation in FreeRTOS

In this work we use FreeRTOS as the operating system to execute both levels of the HSF. FreeRTOS is a portable open source real-time kernel with properties like small and scalable, support for 23 different hardware architectures, and ease to extend and maintain.

The official release of FreeRTOS only supports a single level fixed-priority scheduling. However, we have implemented a two-level HSF for FreeRTOS [26] with associated primitives for hard real-time sharing of resources both within and between servers [27]. The HSF supports reservations by associating a tuple $< Q, P >$ to each server where P is the server period and Q ($0 < Q \leq P$) is the allocated portion of P and is called server budget. Given Q, P, and information on resource holding times, the schedulability of a server and/or a whole system can be calculated with the methods presented in [27].

The HSF implementation supports two kinds of servers, idling periodic [29] and deferrable servers [30]. An idle server is used in the system that has the lowest priority of all the other servers, i.e. 0. It will run when no other server in the system is ready to execute (in idling server). This is useful for maintaining and testing the temporal separation among servers and also useful in testing system behavior.

The implementation uses fixed priority preemptive scheduling (FPPS) for both global and local-level scheduling. The implementation is done with the considerations of being consistent with the underlying operating system, keeping the original API semantics, and doing minimal changes in FreeRTOS kernel to get minimal changes and better utilization of the system.

Server	S1	S2
`Priority`	2	1
`Period`	20	40
`Budget`	10	15

Table 3.1: Servers used to test system behavior.

We have evaluated the performance measures for periodic and deferrable servers on an AVR 32-bit board EVK1100. To test the correctness of the

Tasks	T1	T2	T3
Servers	$S1$	$S1$	$S2$
Priority	1	2	2
Period	20	15	60
Execution Time	4	2	10

Table 3.2: Tasks in both servers.

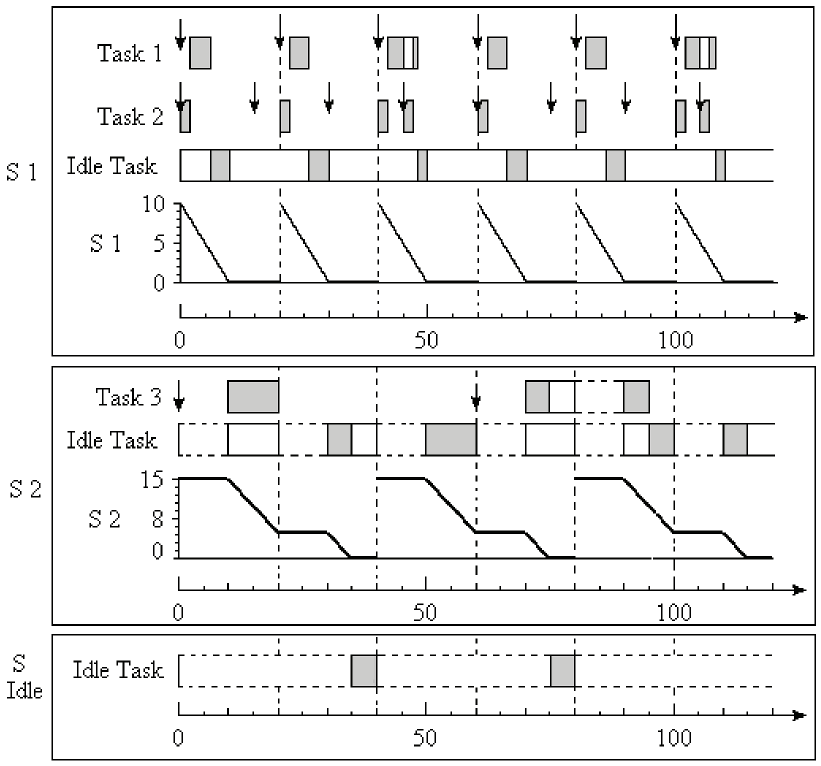


Figure 3.1: Trace for idling periodic servers

server's behavior, the traces of the executions are visualized. The servers used to test the system are given in Table 3.1. Task properties and their assignments to the servers are given in Table 3.2. Note that higher number means higher priority for both servers and tasks.

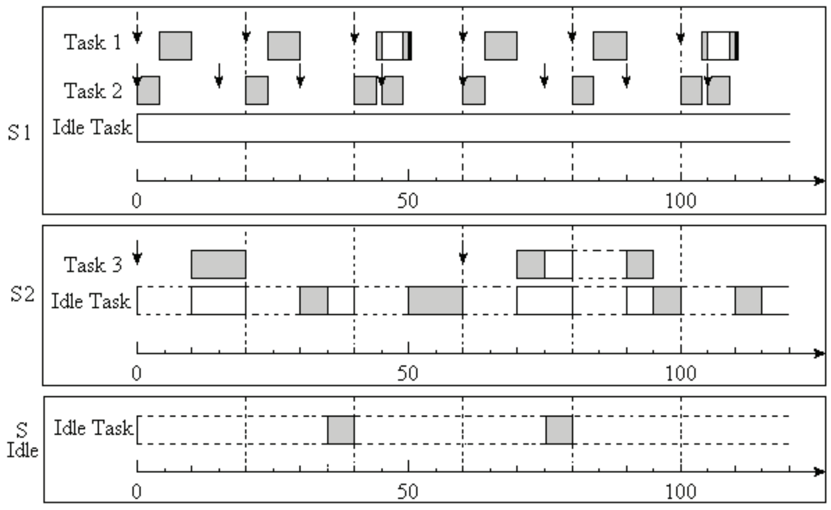


Figure 3.2: Trace showing temporal isolation among idling servers

The servers executions (according to their resource reservations) along with their task sets are presented in Figure 3.1 and Figure 3.2. In these Figures, the horizontal axis represents the execution time starting from 0. In the task's visualization, the arrow represents task arrival and a gray rectangle means task execution. In the server's visualization, the numbers along the vertical axis are the server's capacity, the diagonal line represents the server execution while the horizontal line represents either the waiting time for the next activation (when budget has depleted) or the waiting for its turn to execute (when some other server is executing). Since these are idling periodic servers, all the servers in the system executes till budget depletion, if no task is ready then the idle task of that server executes till its budget depletion.

We have tested the system during normal load, and over loaded conditions. Especially, we tested the system for temporal isolation: The work load of one server was greater than 1, hence its lower priority task was missing its deadline. The same example is executed to perform this test but with the increased utilization of S1. The execution times of T1 and T2 are increased to 4 and 6 respectively, hence making the server S1 utilization greater than 1. Therefore the low priority task T1 misses its deadlines as shown by solid black lines in the Figure 3.2. S1 is never idling because it is overloaded. It is obvious from Figure 3.2, that the overload of S1 does not effect the behavior of S2 even though it has low priority.

The results for behavior testing for the deferrable server are given in Section 6.6.1. We also measure the overhead of the implementation, like tick handler, server context-switch and task context-switch. The detailed results can be found in Section 6.6.2.

For local resource-sharing (within a server) the Stack Resource Policy (SRP) [14] is used, and for global resource-sharing (between servers) the Hierarchical Stack Resource Policy (HSRP) protocol [15] is used with three different overrun mechanisms to deal with the server budget expiration within the critical section: (1) basic overrun without payback (BO), (2) overrun with payback (PO), and (3) enhanced overrun with payback (EO) [16]. The behavior test results for the resource sharing are provided in Section 7.7.1.

3.3 Applying Runnable Virtual Node in ProCom

In ProCom, a virtual node is an integrated model concept. It means that the virtual nodes exist both on the modeling level and on the synthesis level as explained in Section 1.4.1. In the synthesis realm they are called runnable virtual nodes.

3.3.1 Modeling Level

At modeling level, each virtual node contains a set of integrated ProSys components plus the execution resources (a period and budget) required for these ProSys components. The priorities of virtual nodes cannot be assigned at the modeling level. The priorities of a component are relative to other components in the system. Since virtual nodes are developed independently and are meant to be reused in different systems, therefore, the priorities are assigned to virtual nodes later during the synthesis process at the execution level.

3.3.2 Execution Level

At the execution level, the runnable virtual node contains a set of executable tasks, resources required to run those tasks and a real-time local scheduler to schedule these tasks. Note that the runnable virtual node is generated as a result of first deployment step; it is platform independent and not executable as a stand-alone entity.

3.3.3 The Synthesis of the Final Executables

The final executables are generated by assigning priorities to the servers and tasks in the runnable virtual nodes, completing the task bodies with the user code, synthesizing communication among those nodes (if needed) and linking them together with the operating system. These executables then can be downloaded and executed on a physical node. As the real-time properties of runnable virtual nodes are preserved within the servers, therefore when integrated with other runnable virtual nodes on a physical node, the real-time properties of the whole integrated system will be guaranteed by the schedulability analysis of the whole system.

The communication among runnable virtual nodes is provided by a System server, which is automatically generated for inter-node communication (if needed), at this step. The main functionality of the server is to send and receive messages among the nodes. It contains two tasks to achieve this purpose: a sender task and a message-port updater task.

3.4 Evaluation Through a Case Study

The purpose of this case study is to evaluate and demonstrate the execution-time properties and reusability of the run-time components with real-time properties. The PROGRESS Integrated Development Environment (PRIDE) tool [20] supports development of systems using ProCom component models and it has been used for developing the examples of Cruise Controller (CC) and an Adaptive Cruise Controller (ACC) as shown in Figure 3.3.

First, the CC system was realized and exercised to test the temporal isolations among run-time components. Its basic functionality is to keep the vehicle at a constant speed. Then the ACC system extends this functionality by keeping a distance from the vehicle in front by autonomously adapting its speed to the speed of the preceding vehicle and by providing emergency brakes to avoid collisions. To evaluate the reusability of real-time components, the ACC system was realized by the reuse of some runnable virtual nodes from the CC system.

The design, automatic synthesis, and the generation of final binaries are described in details in Section 8.5. In the first step of deployment process,

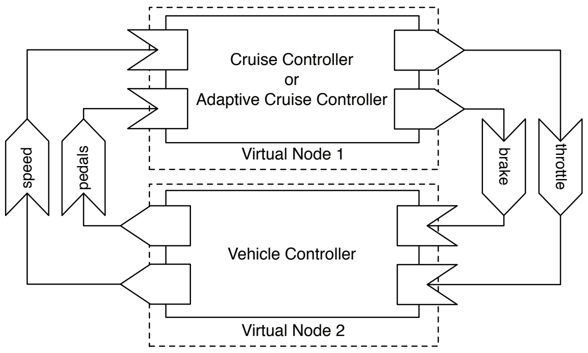


Figure 3.3: Deploying ProSys components on virtual nodes

two runnable virtual nodes are produced for both CC and ACC systems: one runnable virtual node for `Virtual Node1` and one for `Virtual Node2`. These generated nodes contain tasks definitions in them. In the second step of the final synthesis/deployment part, the priorities are assigned to the runnable virtual nodes (also called servers now) and to the tasks in them. Four servers are generated for both examples. In addition to the `system` server and an idle server, two other servers `CC`, and `VC` are used for the CC application while `ACC`, and `VC` are used for the ACC application. In ACC application, the `system` and `VC` servers are reused. The priorities, periods and budgets for these servers are given in Table 3.3.

Server	CC	ACC	VC	SYSTEM
`Priority`	2	2	1	7
`Period`	40	40	60	20
`Budget`	10	10	15	4

Table 3.3: Servers used to test the CC and ACC systems behaviors.

The experiments are performed on the same EVK1100 board using the HSF implementation on FreeRTOS to test both the applications for the temporal isolations and reusability of the runnable virtual node and are provided in Section 8.5.3.

Figure 3.4 demonstrates the system execution under the overload situation to test the temporal isolation among the runnable virtual nodes. The execution times of tasks `CCT1` and `CCT2` are increased by adding the busy loops,

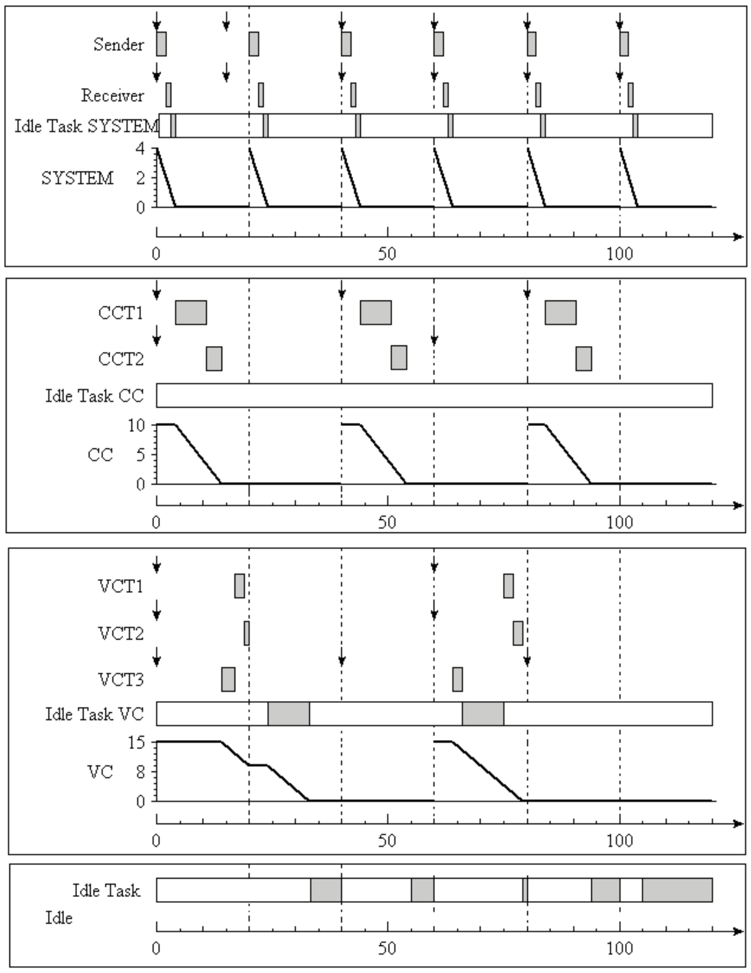


Figure 3.4: Trace showing temporal isolation during overload situation

hence making the `CC` server's utilization greater than 1. Therefore the low priority task `CCT2` misses its deadlines at time 54. The `CC` server is never idling because it is overloaded.

The overload of `CC` server does not effect the behavior of any other server in the system as obvious from Figure 3.4. The `VC` server has a lower prior-

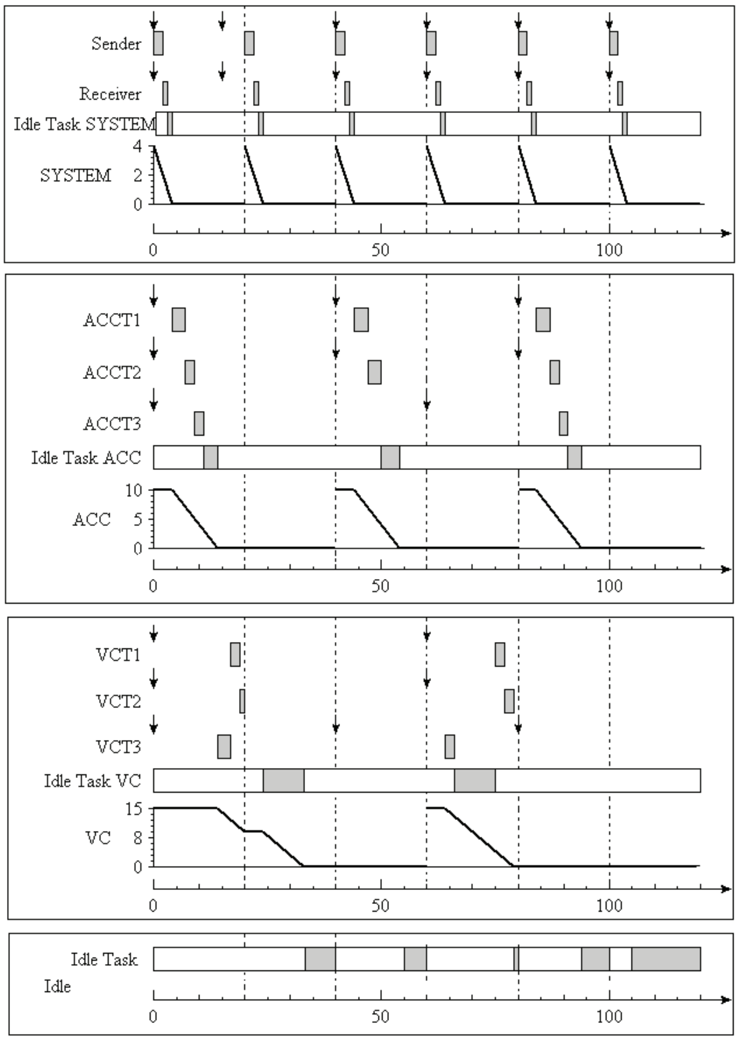


Figure 3.5: Trace showing reusability of runnable virtual nodes in ACC system

ity than the CC, but still it receives its allocated resources and its tasks meet their deadlines. In this manner, the runnable virtual nodes exhibit a predictable timing behavior that eases their integration. It also manifests that the temporal errors are contained within the faulty runnable virtual node only and their effects are not propagated to the other nodes in the system.

Further, to test the reusability of the runnable virtual nodes, the ACC system is synthesized. It also contains four servers: the `ACC` server is synthesized with its task set while the other three servers are reused from the CC system.

The task set for the `ACC` server is different from that of `CC` server. It is clear from the Figure 3.5 that all the three reused servers sustain their timing behavior. For example, the `VC` server has a lower priority than `ACC`, still it's behavior is not effected at all and remains similar to its behavior in the CC system. It confirms the predictable integration of real-time components on one hand, and demonstrates their reusability on the other hand.

We observed the same results on testing the `ACC` server with the changed timing properties, i.e. period 40 and budget 15. As long as the allocated budgets to servers (at the modeling level) are provided, the timing properties are guaranteed at the execution.

Hence, by the use of runnable virtual node components and two-level deployment process, the timing requirements are also encapsulated within the components along with their function requirements and the temporal partitioning is provided among the components (using HSF), that results in the increased predictability during component's integration and making the runnable virtual nodes a reusable entity.

Chapter 4

Conclusions and Future Work

4.1 Summary

In this thesis we have presented an idea of the runnable virtual node using the two-level deployment process to meet the challenges of providing temporal properties, predictable integration and reusability of components with real-time properties. The runnable virtual node uses hierarchical scheduling mechanism to preserve temporal properties within the components as a means to achieve predictable integration and reuse of software in the real-time embedded systems. The runnable virtual node is intended as a coarse grained component for single node deployment and with potential internal multitasking. Each physical node is used to execute one or more virtual nodes.

We have implemented a two-level Hierarchical Scheduling Framework HSF in an open source real-time operating system, FreeRTOS, to support temporal isolation among real-time components. We have implemented idling periodic and deferrable servers using fixed-priority preemptive scheduling at both local and global scheduling levels of HSF. We have focused on being consistent with the underlying operating system and doing minimal changes to get better utilization of the system and kept the original FreeRTOS API semantics. We have added the Stack Resource Policy (SRP) to the FreeRTOS for efficient resource sharing by avoiding deadlocks. Further, we have implemented Hierarchical Stack Resource Policy (HSRP) and overrun mechanisms (without payback, with payback, enhanced overrun) to share global resources in a two-level HFS implementation. We have provided a very simple and easy implementation to execute a legacy system in the HSF with the use of a single API.

We have tested our implementations and performed experimental evaluations on EVK1100 AVR based 32-bit micro-controller. We have validated the imlementations during heavy-load and over-load situations. We have computed the overhead measurements (of tick handler, global scheduler, and task context-switch) and included them into the schedulability analysis.

The notion of two-level deployment process encapsulates the timing properties and uses the hierarchical scheduling within runnable virtual nodes that provides temporal isolation and increases the reuse of the component in different systems. Hence using runnable virtual nodes, a complex embedded system can be developed as a set of well defined reusable components encapsulating functional and timing properties.

Finally, we have performed a proof-of-concept case study which demonstrates temporal error containment within a virtual node as well as reuse of a virtual node in new environment without altering its temporal behavior. Our work is based on the ProCom component-technology running on HSF implementation on FreeRTOS. The case study was executed on an ECU with an AVR based 32-bit micro-controller. However, we believe that our concept is applicable also to commercial component technologies like AADL, AUTOSAR.

4.2 Questions Revisited

In this section we discuss: to which extent the research results and included papers fulfil the goals of Section 2.2. We also comment on the validity of our results.

Paper A addresses research questions Q1, Q2. We found that HSF is a known technique in real-time community that not only provide temporal isolation among subsystems, but also supports isolated and concurrent development of the subsystems. Further in this paper, we propose the idea of virtual node by integrating HSF into component technology to provide temporal isolations among components, that eventually leads to the predictable integration of components. We also present methods to integrate HSF within components and methods to incorporate the idea of virtual node in three component technologies, i.e. ProCom, AUTOSAR, and AADL.

Validity: We explain how to integrate it in only three component models. We cannot claim that the idea of virtual node is applicable in general.

Research question Q3 has a broad scope. We implement HSF in a real-time operating system FreeRTOS; the details are described in Paper B and Paper C. To have an efficient implementation of hierarchical scheduling (less overhead of hierarchical scheduling) and to get better utilization of the system, a number of design considerations are made as explained in Sections 6.4.6 and 7.4.4 and are addressed in Sections 6.5.3 and 7.5.4 respectively. We check our implementation during heavy-load and over-load situations for correctness and efficiency. It is clear from the results of the overhead measurements (of tick handler, global scheduler, and task context-switch) that the design decisions made and the implementations are very efficient.

Validity: We test and validate the implementation by experimental results. Since other existing HSF implementations are either using Linux, VxWorks, or μC/OS-II (using simulator for results), our results are difficult to compare to them. We infer the efficiency of our results on the design decisions and on the implementation done. In this work we have not tried to evaluate different types of HSFs. For that reason we have implemented only two-level fixed-priority scheduling and one resource locking protocol (HSRP).

Paper D is a proof-of-concept paper for the realization of our idea of virtual node in the ProCom component model. It addresses Q4, Q5 by performing an example case study and visualizing the execution traces. We test the system for fault containments, predictability in component's integration and reuse of components in a new environment. Temporal isolation and thus predictable integration has become possible because of the hierarchical scheduling and two-level deployment process. In the experiments, the task sets of each runnable virtual node is executed within the specified budget in a server and is scheduled by a local scheduler. The experimental results manifest that as long as the allocated budgets to servers (at the modeling level) are provided, the timing properties are guaranteed at the execution. Additionaly it also provides fault containments (i.e. temporal errors are contained within the faulty runnable virtual node only and their effects are not propagated to the other nodes in the system). All these properties contribute to the predictability of runnable virtual nodes. The increased predictability during components integration further results in making the runnable virtual nodes a reusable entity as obvious from the results presented in Sections 3.4 (briefly) and 8.5.3 (in details) and Figures 3.4 and 3.5.

Validity: We realize our idea in only ProCom component model. Another limitation is the execution of an example case study (instead of a larger industrial one) due to the immaturity of the PRIDE tool.

4.3 Future Research Directions

This thesis work brings possibilities/issues to conduct further research in certain areas that are not thoroughly addressed and could be interesting to investigate in future. Some of these possibilities could be:

Starting from general issues, in this work we realize our concept of runnable virtual node in the ProCom component model using the PROGRESS Integrated Development Environment (PRIDE) tool by means of a proof-of-concept case study. Currently the PRIDE tool is evolving and the automatic synthesis part is not fully mature. It could be interesting to do some more work on this part and then conduct a larger industrial case study on it.

It could be interesting to support virtual communication-busses using server-based scheduling techniques for e.g. CAN [31] and Ethernet [32] in PRIDE tool. This will allow development, integration and reuse of distributed components using a set of virtual nodes and buses.

We believe that our concept is applicable also to commercial component technologies like AADL, AUTOSAR and it could be interesting to realize the concept in those component technologies too.

In the context of hierarchical scheduling, we assume that a system is executed in a single processor while many real-time applications are executed in a multi-processor or multi-core architecture. It could be interesting to extend the HSF implementation for the multi-processor systems. Further, we only worked on temporal isolation (CPU time division) among different sub-systems in our HSF implementation. Considering the memory isolation issues in the implementation could be another interesting direction.

Another interesting direction could be to make the hierarchical scheduling adaptive in nature by implementing mode switches into the hierarchical scheduling. We would like to start from adapting the CPU time and after that working on the memory issues.

Some of the possibilities to be investigated in future that are specific for each paper are presented in the papers.

Bibliography

[1] Ivica Crnkovic and Magnus Larsson, editors. *Building Reliable Component-Based Software Systems*. Artech House publisher, 2002. ISBN 1-58053-327-2.

[2] Rafia Inam, Jukka Mäki-Turja, Jan Carlson, and Mikael Sjödin. Using temporal isolation to achieve predictable integration of real-time components. In *22nd Euromicro Conference on Real-Time Systems (ECRTS10) WiP Session*, pages 17–20, July 2010.

[3] Rafia Inam, Jukka Mäki-Turja, Jan Carlson, and Mikael Sjödin. Virtual Node – To Achieve Temporal Isolation and Predictable Integration of Real-Time Components. *International Journal on Computing (JoC)*, 1(4), December 2011.

[4] Jan Carlson, Juraj Feljan, Jukka Mäki-Turja, and Mikael Sjödin. Deployment Modelling and Synthesis in a Component Model for Distributed Embedded Systems. In *Proceedings of the 36th Euromicro Conference on Software Engineering and Advanced Applications (SEAA' 10)*, September 2010.

[5] G. Lipari, P. Gai, M. Trimarchi, G. Guidi, and P. Ancilotti. A hierarchical framework for component-based real-time systems. *Component-Based Software engineering*, LNCS-3054(2004):209–216, May 2005.

[6] Z. Deng and J. W.-S. Liu. Scheduling real-time applications in an open environment. In *Proc. 18th IEEE Real-Time Systems Symposium (RTSS)*, 1997.

[7] T.-W. Kuo and C.-H. Li. A fixed-priority-driven open environment for real-time applications. In *Proc. 20th IEEE Real-Time Systems Symposium (RTSS)*, 1999.

[8] G. Lipari and S.Baruah. Efficient scheduling of real-time multi-task applications in dynamic systems. In *Proc. 6th IEEE Real-Time Technology and Applications Symposium (RTAS)*, pages 166–175, 2000.

[9] L. Almeida and P. Pedreiras. Scheduling within temporal partitions: response-time analysis and server design. In *ACM Intl. Conference on Embedded Software(EMSOFT'04)*, pages 95–103, 2004.

[10] X. Feng and A. Mok. A model of hierarchical real-time virtual resources. In *IEEE Real-Time Systems Symposium (RTSS'02)*, pages 26–35, 2002.

[11] R. I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *IEEE Real-Time Systems Symposium (RTSS'05)*, pages 389–398, 2005.

[12] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proc. 24th IEEE Real-Time Systems Symposium (RTSS)*, pages 2–13, 2003.

[13] FreeRTOS web-site. http://www.freertos.org/.

[14] T. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems*, 3(1):67–99, 1991.

[15] R. I. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *IEEE Real-Time Systems Symposium (RTSS'06)*, pages 389–398, 2006.

[16] Moris Behnam, Thomas Nolte, Mikael Sjödin, and Insik Shin. Overrun Methods and Resource Holding Times for Hierarchical Scheduling of Semi-Independent Real-Time Systems. *IEEE Transactions on Industrial Informatics*, 6(1), February 2010.

[17] ATMEL EVK1100 product page. http://www.atmel.com/dyn/Products/tools_card.asp?tool_id=4114.

[18] Tomáš Bureš, Jan Carlson, Ivica Crnković, Séverine Sentilles, and Aneta Vulgarakis. ProCom – the Progress Component Model Reference Manual, version 1.0. Technical Report MDH-MRTC-230/2008-1-SE, Mälardalen University, June 2008.

[19] Séverine Sentilles, Aneta Vulgarakis, Tomáš Bureš, Jan Carlson, and Ivica Crnković. A Component Model for Control-Intensive Distributed Embedded Systems. In *11th International Symposium on Component Based Software Engineering*, pages 310–317, October 2008.

[20] PRIDE Team. PRIDE: the PROGRESS Integrated Development Environment, 2010. "http://www.idt.mdh.se/pride/?id=documentation".

[21] Etienne Borde and Jan Carlson. Towards verified synthesis of procom, a component model for real-time embedded systems. In *14th International ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE)*. ACM, June 2011.

[22] I. Shin and I. Lee. Compositional real-time scheduling framework. In *proceedings of the 25th IEEE International Real-Time Systems Symposium(RTSS'04)*, pages 57–67, 2004.

[23] Z. Deng and J.W.-S. Liu. Scheduling real-time applications in an open environment. In *IEEE Real-Time Systems Symposium(RTSS'97)*, pages 308–319, 1997.

[24] Thomas Nolte, Insik Shin, Moris Behnam, and Mikael Sjödin. A Synchronization Protocol for Temporal Isolation of Software Components in Vehicular Systems. *IEEE Transactions on Industrial Informatics*, 5(4):375–387, November 2009.

[25] Mary Shaw. The Coming-of-Age of Software Architecture Research. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE'01)*, August 2001.

[26] Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, S. M. H. Ashjaei, and Sara Afshar. Hierarchical Scheduling Framework Implementation in FreeRTOS. In *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA' 11)*, pages 1–10, Tolouse, France, September 2011. IEEE Computer Society.

[27] Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, and Moris Behnam. Hard Real-time Support for Hierarchical Scheduling in FreeRTOS. In *7th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT' 11)*, pages 51–60, Porto, Portugal, 2011.

[28] Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, and Jiří Kunčar. Run-Time Component Integration and Reuse in Cyber-Physical Systems. Technical Report MRTC report ISSN 1404-3041 ISRN MDH-MRTC-256/2011-1-SE, Mälardalen University, School of Innovation, Design and Engineering, 2011.

[29] L. Sha, J.P. Lehoczky, and R. Rajkumar. Solutions for some Practical problems in Prioritised Preemptive Scheduling. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, pages 181–191, 1986.

[30] J.K. Strosnider, J.P. Lehoczky, and L. Sha. The deferrable server algorithm for Enhanced Aperiodic Responsiveness in Hard Real-time Environments. *IEEE Transactions on Computers*, 44(1), 1995.

[31] Thomas Nolte, Mikael Nolin, and Hans Hansson. Real-Time Server-Based Communication for CAN. *IEEE Transaction on Industrial Electronics*, 1(3):192–201, April 2005. Citations=33.

[32] Rui Santos, Paulo Pedreiras, Moris Behnam, Thomas Nolte, and Luis Almeida. Hierarchical server-based traffic scheduling in ethernet switches. In *3rd Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS'10)*, pages 69–70, December 2010.

II

Included Papers

Chapter 5

Paper A: Virtual Node – To Achieve Temporal Isolation and Predictable Integration of Real-Time Components

Rafia Inam, Jukka Mäki-Turja, Jan Carlson, Mikael Sjödin
In the Global Science and Technology Forum: International Journal on Computing (JoC), Vol.1, No.4, 2011

Abstract

We present an approach of two-level deployment process for component models used in distributed real-time embedded systems to achieve predictable integration of real-time components. Our main emphasis is on the new concept of virtual node with the use of a hierarchical scheduling technique. Virtual nodes are used as means to achieve predictable integration of software components with real-time requirements. The hierarchical scheduling framework is used to achieve temporal isolation between components (or sets of components). Our approach permits detailed analysis, e.g., with respect to timing, of virtual nodes and this analysis is also reusable with the reuse of virtual nodes. Hence virtual node preserves real-time properties across reuse and integration in different contexts.

Index Terms - Hierarchical scheduling, real-time systems, reusability, component-based software-engineering.

5.1 Introduction

Component integration can be explained as the mechanical task of wiring components together [1]. Since it is rare that two components perfectly match, component integration requires more than just matching the needs and services of one component with the needs and services of others. In real-time embedded systems the components and components integration must satisfy both (1) functional correctness and (2) extra-functional correctness, such as satisfying timing properties.

Temporal behavior of the real-time components poses more difficulties in their integration. When multiple components are deployed on the same hardware node, the timing behavior of each of the components is typically altered in unpredictable ways. This means that a component that is found correct during unit testing may fail, due to a change in temporal behavior, when integrated in a system. Even if a new component is still operating correctly in the system, the integration could cause a previously integrated (and correctly operating) component to fail. Similarly, the temporal behavior of a component is altered if the component is reused in a new system. Since also this alteration is unpredictable, a previously correct component may fail when reused.

Some of these problems can be solved using scheduling analysis [2, 3], however these techniques only allow very simple models; typically simple timing attributes such as period and deadline are used. Components often exhibit a too complex behavior to be amenable for scheduling analysis. And, even if a suitable analysis technique should exist, such analysis requires knowledge of the temporal behavior of all components in the system. Thus, a component cannot be deemed correct without knowing which components it is integrated with. As a result, the reusability of a component is restricted since it is very difficult to know beforehand if the component will pass a schedulability test in a new system.

For large-scale real-time embedded systems, methodologies and techniques are required to provide temporal isolation so that the run-time timing properties could be guaranteed. Further the real-time properties of the components should be maintained for their reuse in large-scale industrial embedded systems.

5.1.1 Contributions

The main contributions of this paper are as follows:

- We propose the concept of a *Virtual Node (VN)*, which is an execution-platform concept that preserves temporal properties of the software executed in the virtual node [4, 5]. The virtual node is intended for coarse-grained components for single node deployment and with potential internal multitasking.

- We propose to *integrate hierarchical scheduling framework (HSF) [6] within the components (virtual nodes)* to realize our ideas of providing temporal properties of real-time components, their predictable integrations and reusability.
- We describe how *the virtual node can be applied in the run-time infrastructure* in three different component technologies: ProCom [7, 8], AUTOSAR [9], and AADL [10].

Outline: Section 5.2 describes the component technologies we study in this paper. In section 5.3, we describe the virtual node execution-mechanism and the hierarchical scheduling framework used by the virtual node. We explain the usage of virtual node in the above mentioned three component models in section 5.4, in section 5.5 we conclude the paper and present the future work to be done.

5.2 Component Technologies

Component-Based Software Engineering (CBSE) and Model-Based Engineering (MBE) are two emerging approaches to develop embedded control systems like software used in trains, airplanes, cars, industrial robots, etc. In this section we briefly outline the component technologies we will target in our work. We discuss three representatives of technologies that use component-based software engineering (AUTOSAR), model-based engineering (AADL) and a combination of CBSE and MBE (ProCom).

We present related work from the perspective of deployment of the components on physical platform and the generation of final executables of the system in the above mentioned technologies.

5.2.1 ProCom

ProCom component model combines both CBSE and MBE techniques for the development of the system parts, hence also exploits the advantages of both. It takes advantages of encapsulation, reusability, and reduced testing from CBSE. From MBE it makes use of automated code generation and performing analysis at an earlier stage of development. In addition ProCom achieves additional benefits of combining both approaches (like flexible reuse, support for mixed maturity, reuse and efficiency tradeoff) [4].

The ProCom component model can be described in two distinct realms: modeling and runnable realms as shown in Figure 5.1. In the modeling realm the models are made using CBSE and MBE, while in the runnable realm the synthesis of runnable entities is done from the model entities. Both realms are explained as follows:

The Modeling Realm

Modeling in ProCom is done by four discrete but related formalisms as shown in Figure 5.1. The first two formalisms relate to the system functionality modeling while the later two represent the deployment modeling of the system.

Functionality of the system is modeled by the ProSave and ProSys components at different levels of granularity. The basic functionality (data and control) of a simple component is captured in ProSave component level, which is passive in nature. At the second formalism level many ProSave components are mapped to make a complete subsystem called ProSys that is active in nature. Both ProSys and ProSave allow composite components. For details on ProSave and ProSys, including the motivation for separating the two, see [7, 8].

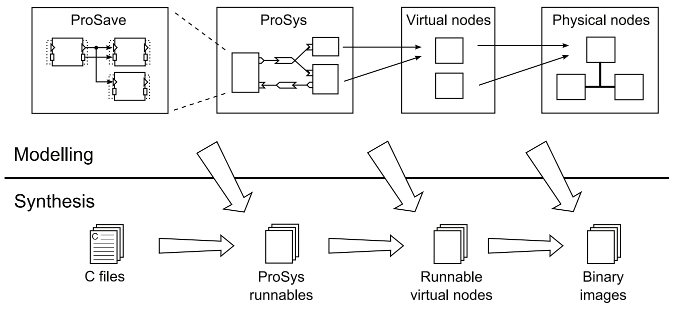


Figure 5.1: The ProCom component model: Overview of the modeling- and runnable realms.

The deployment modeling is used to capture the deployment related design decisions and then mapping the system to run on the physical platform. Many ProSys components can be mapped together on a virtual node (many-to-one mapping) together with a resource budget (i.e. CPU usage and memory requirements) required by those components.

After that many virtual nodes could be mapped on a physical node i.e. an ECU: an electronic control unit. The relationship is again many-to-one. This part represents all the physical nodes, their intercommunication through the network and the type of the network etc. Figure 5.2 represents how four virtual nodes (VN1, VN2, VN3, and VN4) are allocated to the three physical nodes (Node1, Node2, and Node3). Details about the deployment modeling are provided in [4].

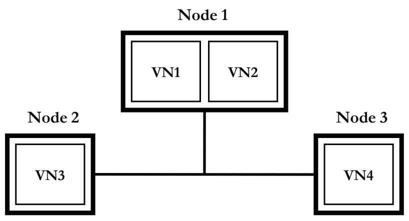


Figure 5.2: Allocation of the virtual nodes to the physical nodes.

The Runnable Realm

is the synthesis of the runnables/executables from the ProCom model entities. The primitive ProSave components are represented as simple C language source code in runnable form. From this C code the ProSys runnables are generated which contains the collection of operating system tasks. Virtual node runnables will implement the local scheduler and will contain the server task. Hence virtual node runnable actually encapsulates the set of tasks, resource allocations, and a real-time scheduler within a server in a two-level hierarchical scheduling framework. Final binary image will be generated by connecting different virtual nodes together with a global scheduler and using some middleware to provide intra-communications among the virtual node executables. As this work is going on, some of the details about the runnable realm are given in [5].

Deployment-Two-steps Process

Rather than deploying a whole system in one big step, the deployment of the ProCom components on the physical platform is done in the following two steps:

1. First the ProSys subsystems are deployed on an intermediate node called Virtual Node. The allocation of ProSys subsystems to the virtual nodes is many-to-one relationship. The additional information that is added at this step is the resource budgets.
2. The virtual nodes are then deployed on the physical nodes. The relationship is again many-to-one means more than one virtual node can be deployed to one physical node.

This two-steps deployment process allows not only the detailed analysis in isolation from the other components to be deployed on the same physical node, but once checked for correctness, it also preserves its temporal properties for further reuse of this virtual node as an independent component. Section 5.3 describes this further.

5.2.2 AUTOSAR

AUTomotive Open System ARchitecture (AUTOSAR) [9] is an open standard for automotive electronics architecture. It is developed by a number of automotive manufacturers and suppliers to deal with the increasing complexity and to fulfill a number of future vehicle requirements (such as safety and availability, driver assistance, software updates, environment, and infotainment). The key features of AUTOSAR are modularity, configurability, standardized interfaces and a runtime environment. It provides standardized modular software infrastructure and basic software for embedded automotive systems. A layered-software platform has been developed to achieve modularity, scalability, transferability, and reusability of components. AUTOSAR methodology is a standardized technique that describes all the major steps in a complete development cycle of a system. It encloses all steps from the system level configurations till the generation of ECU executable binaries. Functional software is developed using component-based approach. A component is developed over many layers of AUTOSAR, including: Application layer, Runtime Environment (RTE), Basic software and ECU hardware as shown in Figure 5.3. Some important layers are:

- Application layer resides at the top of RTE. At this layer, an application consists of one or many AUTOSAR software components and sensor/actuator components.
- RTE connects AUTOSAR components. It is responsible for configurations and communication among components. It enables both communi-

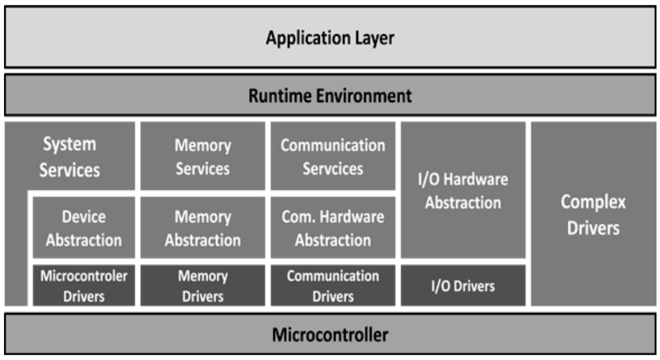


Figure 5.3: AUTOSAR layered architecture [9].

cation between components on the same ECU and also communication between components on different ECUs. Hence it makes the components completely independent from the underlying hardware. Components communicate with each other using ports (e.g., PPort, RPort) and port interfaces (e.g., client-server, sender-receiver).

- Basic software (BSW) provides services to Input/ Output (I/O), communication, memory, and system. It has access to hardware (e.g., sensors, actuators), Internal/External memory, microcontroller onboard peripheral devices and communications. BSW consists of Internal drivers (e.g., EEPROM, CAN, etc.), external drivers (e.g., external EEPROM, etc.), Interfaces that offer generic API for upper layers, handlers, and managers. BSW uses complex drivers to handle timing and functional requirements of complex sensors and actuators.

- Microcontroller Abstraction layer resides at the bottom just above the underlying ECUs. It separates the above layers from the hardware and provides standardized interfaces for communication of upper layers to the ECU.

Software component (SW-C) at ECU level contains at least one or several runnable entities (or simply runnables). A runnable is small fragment of sequential code within a component. Runnable entities are grouped into operating system tasks executed on ECUs. Runnables grouped onto one task may belong to different software components. Operating system controls and schedules these tasks. These OS tasks can be of one of the categories, basic tasks (Category1 without WaitEvent) or extended tasks (Category2 with WaitEvent). All runnables are activated by RTEEvents.

Deployment

Deployment in AUTOSAR begins when RTE generator maps all runnables to the OS tasks and build inter-ECU and intra-ECU communications among them. This mapping is dependent on different extra-functional properties and behaviors of the runnables e.g., runnable with Category1 will be mapped differently from the runnable with Category2. Three different rules for mapping are given in the AUTOSAR RTE specifications [11]. After mapping, RTE generator configures each ECU. In the last, the OS tasks bodies are constructed by RTE generator.

The main disadvantage of AUTOSAR is that it lacks clear and well-defined timing properties that further affect the execution semantics too. A tool suite supporting the complete AUTOSAR methodology is still missing.

5.2.3 AADL

Architecture Analysis and Design Language (AADL) was developed as a SAE Standard AS-5506 [10] in 2004 to design and analyze software and hardware architectures of distributed real-time embedded systems. It supports MBE and has both textual and graphical representations. It also supports syntax and semantics analyses of the language. Modeling of software and hardware parts is supported by software components (e.g., process, data, thread, thread group, subprogram), and execution platform components (e.g., processor, memory, bus, device) respectively. It also allows hybrid components (e.g., system) [12]. Properties and new functional aspects can be attached to the elements (e.g., components, connections) using the properties defined in the SAE standard, and communication among components is performed using component interfaces i.e., ports. Ocarina [12] is a tool suite by Telecom Paris that facilitates the design of AADL models and their mapping on a hardware platform, assessment of these models (e.g., syntactic/semantic analysis, schedulability analysis performed by Ocarina and Cheddar [13]), and then automatic code generation from these models and their deployment.

Deployment

Automatic code generation is done using the Ocarina compiler [12] that comprises of two traditional parts: the frontend and the backend.

1. *The frontend* is responsible for lexical checking, syntactic analysis, semantic analysis and instantiation. It generates the lexems, then generates the abstract syntax tree and add semantics to it, and at the last step produces the instance tree. It also scrutinizes all syntactic, semantic and instance errors and warnings.

2. *The backend* part is responsible for code generation in three steps; first the expansion of instance tree, second the conversion of this instance tree into a syntactic tree of the target language (Ada or C) and the last step is the code generation that generates the code in C or Ada language.

Ocarina supports code generation in Ada and C languages using a middleware API called PolyORB (PolyORB for Ada while PolyORB-HI for C). This middleware provides execution services and wraps the POSIX API, hence it is POSIX compliant. Runnable entities are presented by processes. A process contains many tasks and it is a selfcontained runnable entity that executes on a hardware platform without any programmatic dependencies. The final executable binaries are generated by compiling the Ocarina automatic generated code (in C or Ada) together with the user written application code (in C or Ada) and the AADL runtime (e.g. PolyORB, PolyORB-HI).

POK is another type of runnable entity for AADL is implemented by Julien [14]. This technique is an extension of the first one implemented by Ocarina. It employs a hierarchical scheduling concept in a partition. A partition is a combination of several processes and a scheduler called Virtual Processor. Each partition is isolated in terms of space and time. Each process again encloses several tasks and a local scheduler. A local scheduler schedules all the tasks of a particular process. Virtual Processor is then responsible for scheduling all the processes in a particular partition.

5.3 Virtual Node

The concept of virtual node is used to achieve not only temporal isolation and predictable temporal properties of real-time components but also to get better reusability of components with real-time properties. Further it reduces the efforts related to testing, validation and certification. This concept is based on two-level deployment process. It means that the whole system is generated in two steps rather than a big synthesis step. At the first level of deployment, functionality (in form of design-time components) is deployed to virtual nodes, and virtual nodes are assigned execution resources. In this way behavior is encapsulated with respect to timing and resource usage and VN becomes a reusable component in addition to the design-time components. In the second level of deployment, these virtual nodes are deployed on a physical platform together with a global scheduler [5].

A virtual node includes the executable representation of the components (e.g. a set of tasks), a resource allocation, and a real-time scheduler to be executed within a server in the hierarchical scheduling framework. Hierarchical scheduling is described as follow:

5.3.1 Hierarchical Scheduling Framework (HSF)

A two-level Hierarchical Scheduling Framework (HSF) [6] is used to provide the temporal isolation among the virtual nodes. In hierarchical scheduling, the CPU is partitioned into a set of servers, each server can use a different scheduling policy, and are in turn scheduled by a global (system-level) scheduler. Hence a two-level HSF can be viewed as a tree with one parent node (global scheduler) and many leaf nodes (local schedulers) as illustrated in Figure 5.4.

The leaf nodes contain its own internal set of tasks that are scheduled by a local (subsystem-level) scheduler. The parent node is a global scheduler that schedules local schedulers. Using an appropriate HSF, subsystems can be developed and analyzed in isolation from each other. As each subsystem has its own local scheduler, after satisfying the temporal constraints, the temporal properties are saved within each subsystem. Later, a global scheduler is used to combine all the subsystems together without violating the temporal constraints that are already analyzed and stored in them. Accordingly we can say that the HSF provides partitioning of the CPU between different servers. Thus, server-functionality can be isolated from each other for, e.g., fault containment, compositional verification, validation and certification, and unit testing.

Using HSF a subsystem (virtual node in our case) can be developed and analyzed in isolation, with its own local scheduler at first step of deployment and its temporal properties are preserved. Then at the second step of deployment an arbitrary global scheduler is used for the integration of multiple subsystems (virtual nodes) without violating the temporal properties of the individual subsystems analyzed in isolation. A brief overview of our hierarchical scheduling

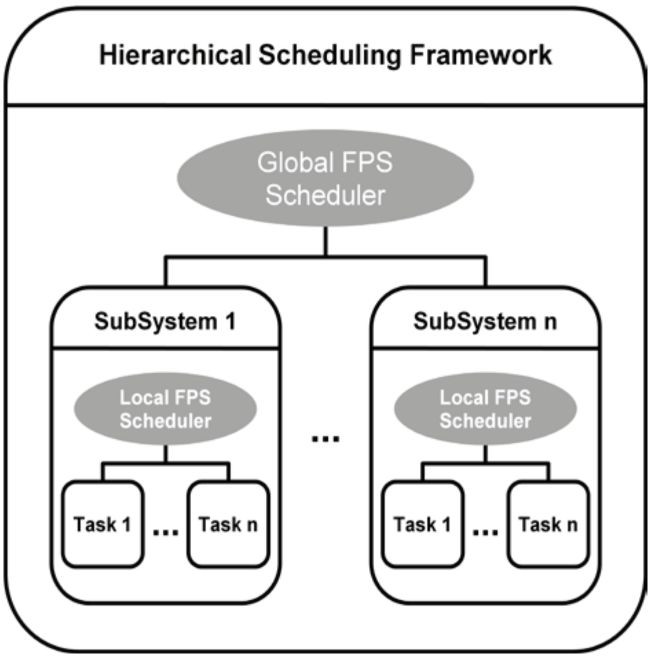


Figure 5.4: Two-level hierarchical scheduling framework.

framework implementation is given here.

5.3.2 HSF Implementation in FreeRTOS

The two-level hierarchical scheduling implementation is done independently from components [15, 16]. We have chosen FreeRTOS [17], a portable open source real-time scheduler for the implementation. Its main properties like open source, small and scalable, support for 23 different hardware architectures, and ease to extend and maintain makes it a perfect choice to be used within the PROGRESS project [7, 8]. The motivations for choosing FreeRTOS and the details about its real-time kernel are provided in [15, 16].

We have implemented time-triggered periodic tasks within the FreeRTOS operating system to support hard real-time components. The HSF implementation supports two kinds of servers, idling periodic and deferrable servers.

The implementation uses fixed priority preemptive scheduling (FPPS) for both global and local-level scheduling. FPPS is flexible and simple to implement, plus is the de-facto industrial standard for task scheduling and FreeRTOS native scheduling policy. The resource sharing policy of FreeRTOS to access local shared resources has been improved, and the support for inter-subsystem resource sharing to access global shared resources has been implemented in the HSF implementation. This entails: support for Stack Resource Policy (SRP) [18] for local resource sharing to avoid problems like priority inversions and deadlocks, and Hierarchical Stack Resource Policy (HSRP) [19] for global resource sharing with three different methods to handle overrun [20] to handle the budget expiration within the critical section. These three types of overrun mechanisms are overrun without payback (BO), with payback (PO), and enhanced overrun (EO). Implementation of BO is very simple, the server simply executes and overruns its budget, and no further action is required. For PO and EO we need to measure the overrun amount of time to pay back at the server's next activation. We have also provided legacy support for existing systems or components to be executed within our HSF implementation as a subsystem.

We have performed a detailed experimental evaluation [15, 16] on the implementation to test its temporal behavior and performance measures on an AVR-based 32-bit EVK1100 board [21]. The AVR32UC3A0512 micro-controller runs at the frequency of 12MHz and its tick interrupt handler at 1ms. We have tested the implementation for the correct behavior of idling and deferrable servers and of overrun mechanisms by plotting the traces of the execution of the system. We have also evaluated the system behavior during the overload situation and tested the temporal isolation among servers. We showed that when one server is overloaded and its tasks miss deadlines, it does not affect the behavior of other servers in the system, even if the priority of the overloaded server is highest; hence proves the temporal isolation and fault containment behavior of HSF.

5.4 Applying Virtual Node Concept to ProCom, AUTOSAR, and AADL

In the component models we are currently studying the virtual node concept to be applied in the following way:

5.4.1 ProCom

In ProCom the Virtual Node is an integrated model concept. That means that the virtual nodes exist both on the modeling level and as executable entities as shown in Figure 5.1. The system is generated using two-level deployment process rather than a big synthesis step. A set of ProSys subsystems are mapped to one virtual node which can then be integration-tested and validated for the correct temporal behavior. This virtual node preserves its temporal properties and hence becomes a reusable entity that is ready to deploy in numerous systems and stored for future reuse.

At the modeling level, each virtual node contains a set of integrated ProSys components plus the resources (CPU budget, memory) required for these ProSys components. At the executable level, virtual node contains the set of executable tasks, resources required to run those tasks and a real-time local scheduler to schedule these tasks. The local scheduler runs within a global scheduler in a HSF.

The final executables that can be downloaded and executed on the physical node consists of a set of virtual nodes and simple real-time scheduler linked together. The scheduler is the top level scheduler in the hierarchical scheduling framework, and is responsible for dispatching the servers of each virtual node according to their bandwidth reservation. As the real-time properties of the virtual node are preserved within the local scheduler, therefore when integrated with other virtual nodes on a physical node, the real-time properties of the whole integrated system will be guaranteed.

5.4.2 AUTOSAR

For AUTOSAR, we propose to map a number runnables to a virtual node. Thus, an AUTOSAR component can be deployed to a set of virtual nodes; the natural choice would be to use one virtual node per physical node that the component will be distributed over. Using this approach the component can be developed and its timing behavior tested without accounting for interference from other AUTOSAR components deployed at the same physical nodes.

However, since the AUTOSAR component-model and methodology does not recognize the virtual node as an entity of its own, reuse in different organizations or different software architectures may be difficult. However, the virtual node still provides strong encapsulation of the runnables and thus makes the functionality robust against future changes in both the runnables and in other components running in other virtual nodes.

5.4.3 AADL

For AADL, we propose to map the generated code from AADL models along with user written code to the virtual node. Hence instead of synthesizing whole system in a single big-bang step, the synthesis will be performed in smaller steps. The synthesis will be done at the two levels:

1. First the individual runnables will be created in isolation and timing analysis will be performed on them.
2. Then some middleware (e.g., PolyORB, PolyORB-HI) can be used for their intra-communications and to generate a whole system.

Currently a similar concept of two level code generation has been used for ARINC653 systems [22] using AADL. It is supported by the tool POK [14] that uses Ocarina tool for AADL models development and Cheddar tool for scheduling analysis. POK supports partitioning of the CPU and hierarchical scheduling for the underlying ARINC653 systems by using virtual processor. This approach is not generic in embedded real-time systems since ARINC653 is an avionics standard, therefore, the use of virtual processor is restricted to the avionics only.

5.5 Conclusions and Future Work

We have described our technique of two-level deployment process to allow predictable integration of software components with temporal requirements. The technique is based on the concept of virtual nodes which use hierarchical scheduling to achieve temporal isolation and predictable execution of components allocated to the virtual nodes. The virtual node will become a real-time executable reusable entity. We have described how this technique can be used for three different component models: ProCom, AUTOSAR and AADL.

Future work is to do the code synthesis for generating and configuring virtual nodes from ProSys subsystems in ProCom component model. It includes the integration of our HSF implementation within the virtual node. Once these implementation efforts are complete will have all the links in a complete development chain for model driven engineering of component based system in the ProCom component technology:

- Using the ProCom Integrated Development Environment (PRIDE) components can be developed, assembled and deployed to virtual nodes.

- Using scheduling analysis of hierarchically scheduled systems [20] we can determine schedulability of both individual virtual nodes and the final composition of multiple virtual nodes on a single physical node.
- And, with our implemented code synthesis and runtime platform we can generate and execute the components and their applications in a predictable way.

The next step will then be to validate the generality of the virtual-node concept by applying it to AUTOSAR and AADL technologies.

Bibliography

[1] Ivica Crnkovic and Magnus Larsson, editors. *Building Reliable Component-Based Software Systems*. Artech House publisher, 2002. ISBN 1-58053-327-2.

[2] L. Sha, T. Abdelzaher, K-E. rzn, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real Time Scheduling Theory: A Historical Perspective. *Real-Time Systems*, 28(2/3):101–155, 2004.

[3] J.A. Stankovic, M. Spuri, M. Di Natale, and G.C. Buttazzo. Implications of Classical Scheduling Results for Real-Time Systems. *IEEE Computer*, pages 16–25, June 1995.

[4] Jan Carlson, Juraj Feljan, Jukka Mäki-Turja, and Mikael Sjödin. Deployment Modelling and Synthesis in a Component Model for Distributed Embedded Systems. In *Proceedings of the 36th Euromicro Conference on Software Engineering and Advanced Applications (SEAA' 10)*, September 2010.

[5] Rafia Inam, Jukka Mäki-Turja, Jan Carlson, and Mikael Sjödin. Using temporal isolation to achieve predictable integration of real-time components. In *22nd Euromicro Conference on Real-Time Systems (ECRTS10) WiP Session*, pages 17–20, July 2010.

[6] Z. Deng and J.W.-S. Liu. Scheduling real-time applications in an open environment. In *IEEE Real-Time Systems Symposium(RTSS'97)*, pages 308–319, 1997.

[7] Tomáš Bureš, Jan Carlson, Ivica Crnković, Séverine Sentilles, and Aneta Vulgarakis. ProCom – the Progress Component Model Refer-

ence Manual, version 1.0. Technical Report MDH-MRTC-230/2008-1-SE, Mälardalen University, June 2008.

[8] Tomáš Bureš, Jan Carlson, Séverine Sentilles, and Aneta Vulgarakis. A Component Model Family for Vehicular Embedded Systems. In *The 3rd International Conference on Software Engineering Advances*. IEEE, October 2008.

[9] Autosar project-page. www.autosar.org.

[10] SAE International. AADL specification. http://www.sae.org/.

[11] AUTOSAR Partnership. Specification of RTE V2.0.1 R3.0 Rev 0001 , 2008. http://www.autosar.org/.

[12] G. Lasnier, B. Zalila, L. Pautet, and J. Hugues. *OCARINA : An Environment for AADL Models Analysis and Automatic Code Generation for High Integrity Applications*. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-01923-4.

[13] F. Singhoff, J. Legrand, L. Nana, and L. Marc. Cheddar: a flexible real time scheduling framework. *Ada Lett.*, XXIV(4):1–8, 2004.

[14] Julien Delange, L. Pautet, A. Plantec, M. Kerboeuf, F. Singhoff, and F. Kordon. Validate, simulate, and implement arinc653 systems using the aadl. *Ada Lett.*, 29(3):31–44, 2009.

[15] Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, S. M. H. Ashjaei, and Sara Afshar. Hierarchical Scheduling Framework Implementation in FreeRTOS. In *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA' 11)*, pages 1–10, Tolouse, France, September 2011. IEEE Computer Society.

[16] Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, and Moris Behnam. Hard Real-time Support for Hierarchical Scheduling in FreeRTOS. In *7th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT' 11)*, pages 51–60, Porto, Portugal, 2011.

[17] FreeRTOS web-site. http://www.freertos.org/.

[18] T. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems*, 3(1):67–99, 1991.

[19] R. I. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *IEEE Real-Time Systems Symposium (RTSS'06)*, pages 389–398, 2006.

[20] Moris Behnam, Thomas Nolte, Mikael Sjödin, and Insik Shin. Overrun Methods and Resource Holding Times for Hierarchical Scheduling of Semi-Independent Real-Time Systems. *IEEE Transactions on Industrial Informatics*, 6(1), February 2010.

[21] ATMEL EVK1100 product page. http://www.atmel.com/dyn/Products/tools_card.asp?tool_id=4114.

[22] Airlines Electronic Engineering. Avionics Application Software Standard Interface. Technical report, Aeronautical Radio, INC, 1997.

Chapter 6

Paper B: Support for Hierarchical Scheduling in FreeRTOS

Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, Seyed Mohammad Hossein Ashjaei, Sara Afshar

In Proceedings of the 16^{th} IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 11), pages 1-10, IEEE Industrial Electronics Society, Toulouse, France, September, 2011.
Awarded scholarship by IEEE Industrial Electronic Society as best student paper.

Abstract

This paper presents the implementation of a Hierarchical Scheduling Framework (HSF) on an open source real-time operating system (FreeRTOS) to support the temporal isolation between a number of applications, on a single processor. The goal is to achieve predictable integration and reusability of independently developed components or applications. We present the initial results of the HSF implementation by running it on an AVR 32-bit board EVK1100.

The paper addresses the fixed-priority preemptive scheduling at both global and local scheduling levels. It describes the detailed design of HSF with the emphasis of doing minimal changes to the underlying FreeRTOS kernel and keeping its API intact. Finally it provides (and compares) the results for the performance measures of idling and deferrable servers with respect to the overhead of the implementation.

6.1 Introduction

In real-time embedded systems, the components and component integration must satisfy both (1) functional correctness and (2) extra-functional correctness, such as satisfying timing properties. Temporal behavior of real-time components poses more difficulties in their integration. The scheduling analysis [1, 2] can be used to solve some of these problems, however these techniques only allow very simple models; typically simple timing attributes such as period and deadline are used. In addition, for large-scale real-time embedded systems, methodologies and techniques are required to provide not only spatial isolation but also temporal isolation so that the run-time timing properties could be guaranteed.

The Hierarchical Scheduling Framework (HSF) [3] is a promising technique for integrating complex real-time components on a single processor to overcome these deficiencies. It supplies an efficient mechanism to provide temporal partitioning among components and supports independent development and analysis of real-time systems. In HSF, the CPU is partitioned into a number of subsystems. Each subsystem contains a set of tasks which typically would implement an application or a set of components. Each task is mapped to a subsystem that contains a local scheduler to schedule the internal tasks of the subsystem. Each subsystem can use a different scheduling policy, and is scheduled by a global (system-level) scheduler.

We have chosen FreeRTOS [4] (a portable open source real-time scheduler) to implement hierarchical scheduling framework. Its main properties like open source, small footprint, scalable, extensive support for different hardware architectures, and easily extendable and maintainable, makes it a perfect choice to be used within the PROGRESS project [5].

6.1.1 Contributions

The main contributions of this paper are as follows:

- We have provided *a two-level hierarchical scheduling support* for FreeRTOS. We provide the support for a fixed-priority preemptive global scheduler used to schedule the servers and the support for idling and deferrable servers, using fixed-priority preemptive scheduling.
- We describe the *detailed design* of our implementation with the considerations of doing minimal changes in FreeRTOS kernel and keeping the original API semantics.

- We have *evaluated the performance measures* for periodic and deferrable servers on an AVR 32-bit board EVK1100 [6]. We also measure the overhead of the implementation, like tick handler, server context-switch and task context-switch.

6.1.2 The Hierarchical Scheduling Framework

A two-level HSF [7] can be viewed as a tree with one parent node (global scheduler) and many leaf nodes (local schedulers) as illustrated in Figure 6.1. A leaf node contains its own internal set of tasks that are scheduled by a local (subsystem-level) scheduler. The parent node is a global scheduler and is responsible for dispatching the servers according to their bandwidth reservation. A major benefit of HSF is that subsystems can be developed and analyzed in isolation from each other [8]. As each subsystem has its own local scheduler, after satisfying the temporal constraints of the subsystem, the temporal properties are saved within each subsystem. Later, the global scheduler is used to combine all the subsystems together without violating the temporal constraints that are already analyzed and stored in them. Accordingly we can say that the HSF provides partitioning of the CPU between different servers. Thus, server-functionality can be isolated from each other for, e.g., fault containment, compositional verification, validation and certification, and unit testing.
Outline: Section 6.2 presents the related work on hierarchical scheduler and its implementations. In section 6.3 we provide our system model. Section 6.4 gives an overview of FreeRTOS and the requirements to be incorporated into our design of HSF. We explain the implementation details of fixed-priority servers and hierarchical scheduler in section 6.5. In section 6.6 we test the behavior and evaluate the performance of our implementation, and in section 6.7 we conclude the paper. We provide the API of our implementation in Appendix 6.8.

6.2 Related Work

6.2.1 Hierarchical Scheduling

HSF has attained a substantial importance since introduced in 1990 by Deng and Liu [3]. Numerous studies has been performed for the schedulability analysis of HSFs [9, 10] and processor models [11, 12, 13, 8] for independent subsystems. The main focus of this research has been on the schedulability analysis and not much work has been done to implement these theories.

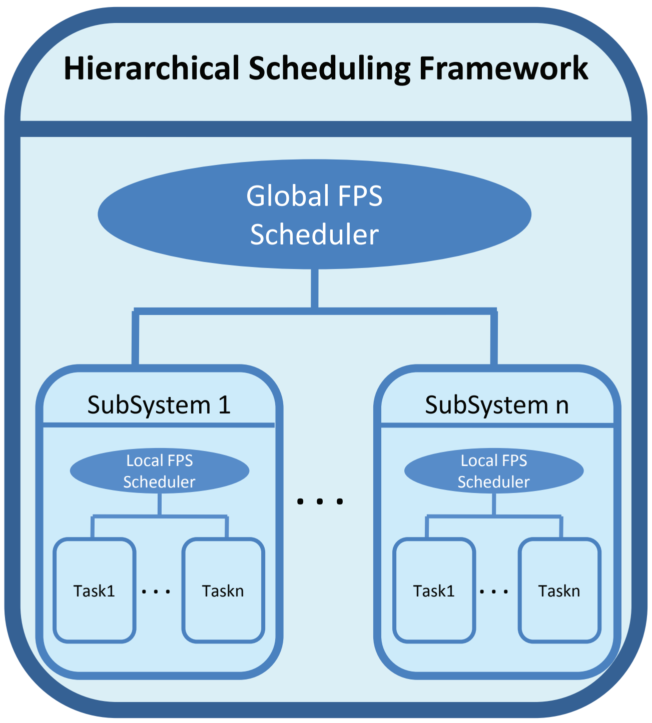


Figure 6.1: Two-level Hierarchical Scheduling Framework

6.2.2 Implementations of Hierarchical Scheduling Framework

Saewong and Rajkumar [14] implemented and analyzed HSF in CMU's Linux/RK with deferrable and sporadic servers using hierarchical deadline monotonic scheduling.

Buttazzo and Gai [15] present an HSF implementation based on Implicit Circular Timer Overflow Handler (ICTOH) using EDF scheduling for an open source RTOS, ERIKA Enterprise kernel.

A micro kernel called SPIRIT-μKernel is proposed by Kim *et al.* [7] based on two-level hierarchical scheduling. They also demonstrate the concept, by porting two different application level RTOS, VxWorks and eCos, on top of the SPIRIT-μKernel. The main focus is on providing a software platform for strongly partitioned real-time systems and lowering the overheads of kernel.

It uses an offline scheduler at global level and the fixed-priority scheduling at local level to schedule the partitions and tasks respectively.

Behnam *et al.* [16] present an implementation of a two-level HSF in a commercial operating system VxWorks with the emphasis on not modifying the underlying kernel. The implementation supports both FPS and EDF at both global and local level of scheduling and a one-shot timer is used to trigger schedulers. The work presented in this paper is different from that of [16]. Our implementation aims at efficiency while modifying the kernel with the consideration of being consistent with the FreeRTOS API.

More recently, Holenderski *et al.* [17] implemented a two-level fixed priority HSF in μC/OS-II, a commercial real-time operating system. This implementation is based on Relative Timed Event Queues (RELTEQ) [18] and virtual timers [19] and stopwatch queues on the top of RELTEQ to trigger timed events. They incorporated RELTEQ queues, virtual timers, and stopwatch queues within the operating system kernel and provided interfaces for it. Their HSF implementation uses these interfaces. Our implementation is different from that of [17] in the sense that we only extend the functionality of the operating system by providing support for HSF, and not changing or modifying the data structures used by the underlying kernel. We aim at efficiency, simplicity in design, and understandability and keeping the FreeRTOS original API intact. Also our queue management is very efficient and simple that eventually reduces the overhead.

6.3 System Model

In this paper, we consider a two-level hierarchical scheduling framework, in which a global scheduler schedules a system S that consists of a set of independently developed subsystems S_s, where each subsystem S_s consists of a local scheduler along with a set of tasks.

6.3.1 Subsystem Model

Our subsystem model conforms to the periodic processor resource model proposed by Shin and Lee [8]. Each subsystem S_s, also called server, is specified by a subsystem *timing interface* $S_s(P_s, Q_s)$, where P_s is the period for that subsystem $(P_s > 0)$, and Q_s is the capacity allocated periodically to the subsystem $(0 < Q_s \leq P_s)$. At any point in time, B_s represents the remaining budget during the runtime of subsystem. During execution of a subsystem, B_s

is decremented by one at every time unit until it depletes. If $B_s = 0$, the budget is depleted and S_s will be suspended until its next period where B_s is replenished with Q_s. Each server S_s has a unique priority p_s. There are 8 different subsystem priorities (from lowest priority 1 to the highest 7). Only idle server has priority 0. In the rest of this paper, we use the term subsystem and server interchangeably.

6.3.2 Task Model

In the current implementation, we use a very simple task model, where each task τ_i is characterized only by its priority ρ_i. A task, τ_i has a higher priority than another task, τ_j, if $\rho_i > \rho_j$. There can be 256 different task priorities, from lowest priority 1 (only idle task has priority 0) to the highest 255.

The local-level resource sharing among tasks of the same subsystem uses the FreeRTOS resource sharing methods. For the global-level resource sharing user should use some traditional waitfree [20] technique.

6.3.3 Scheduling Policy

We use a fixed-priority scheduling, FPS, at both the global and the local levels. FPS is the native scheduling of FreeRTOS, and also the predominant scheduling policy used in embedded systems industry. We use the First In First Out, FIFO, mechanism to schedule servers and tasks under FPS when they have equal priorities.

6.4 FreeRTOS

6.4.1 Background

FreeRTOS is a portable, open source (licensed under a modified GPL), mini real-time operating system developed by Real Time Engineers Ltd. It is ported to 23 hardware architectures ranging from 8-bit to 32-bit micro-controllers, and supports many development tools. Its main advantages are portability, scalability and simplicity. The core kernel is simple and small, consisting of three or four (depends on the usage of coroutines) C files with a few assembler functions, with a binary image between 4 to 9KB.

Since most of the source code is in C language, it is readable, portable, and easily expandable and maintainable. Features like ease of use and understandability makes it very popular. More than $77,500$ official downloads in

2009 [21], and the survey result performed by professional engineers in 2010 puts the FreeRTOS at the top for the question ”which kernel are you considering using this year” [22] showing its increasing popularity.

The FreeRTOS kernel supports preemptive, cooperative, and hybrid scheduling. In the fixed-priority preemptive scheduling, tasks with the same priority are scheduled using the Round-Robin (RR) policy. It supports any number of tasks and very efficient context-switching. FreeRTOS supports both static and dynamic (changed at run-time) priorities of the tasks. It has binary, counting and recursive semaphores and the mutexes for resource protection and synchronization, and queues for message passing among tasks. Its scheduler runs at the rate of one tick per milli-second by default, but it can be changed to any other value easily by setting the value of `configTICK_RATE_HZ` in the FreeRTOSConfig.h file.

We have extended FreeRTOS with a two-level hierarchical scheduling framework. The implementation is made under consideration of not changing the underlying operating system kernel unless vital and keeping the semantics of the original API. Hence the hierarchical scheduling of tasks is implemented with intention of doing as few modifications to the FreeRTOS kernel as possible.

6.4.2 Support for FIFO Mechanism for Local Scheduling

Like many other real-time operating systems, FreeRTOS uses round robin scheduling for tasks with equal priorities. FreeRTOS uses `listGET_OWNER_OF_NEXT_ENTRY` macro to get the next task from the list to execute them in RR fashion. We change it to the FIFO policy to schedule tasks at local-level. We use `listGET_ OWNER_OF_HEAD_ENTRY` macro to execute the current task until its completion. At global-level the servers are also scheduled using the FIFO policy.

6.4.3 Support for Servers

In this paper we implement the idling periodic [23] and deferrable servers [24]. We need periodic activation of local servers to follow the periodic resource model [8]. To implement periodic activation of local servers our servers behave like periodic tasks, i.e. they replenish their budget Q_s every constant period P_s. A higher priority server can preempt and the execution of lower priority servers.

Support for Idling Periodic Server

In the idling periodic server, the tasks execute and use the server's capacity until it is depleted. If server has the capacity but there is no task ready then it simply idles away its budget until a task becomes ready or the budget depletes. If a task arrives before the budget depletion, it will be served. One idle task per server is used to run when no other task is ready.

Support for Deferrable Server

In the deferrable server, the tasks execute and use the server's capacity until it is depleted. If the server has capacity left but there is no task ready then it suspends its execution and preserves its remaining budget until its period ends. If a task arrives later before the end of server's period, it will be served and consumes server's capacity until the capacity depletes or the server's period ends. If the capacity is not used till the period end, then it is lost. In case there is no task (of any server) ready in the whole system, an idle server with an idle task will run instead.

Support for Idle Server

When there is no other server in the system to execute, then an idle server will run. It has the lowest priority of all the other servers, i.e. 0. It contains only an idle task to execute.

6.4.4 System Interfaces

We have designed the API with the consideration of being consistent in structure and naming with the original API of FreeRTOS.

Server Interface

A server is created using the function `vServerCreate(period, budget, priority, *serverHandle)`. A macro is used to specify the server type as idling periodic or deferrable server in the config file.

Task Interface

A task is created and assigned to the specific server by using the function `xServerTaskCreate()`. In addition to the usual task parameters passed to create a task in FreeRTOS, a handle to the server `serverHandle` is passed to this function to register the newly created task to its parent server. The original FreeRTOS API to create the task cannot be used in HSF.

6.4.5 Terminology

The following terms are used in this paper:

- **Active servers:** Those servers whose remaining budget (B_s) is greater than zero. They are in the ready-server list.

- **Inactive servers:** Those servers whose budget has been depleted and waiting for their next activation when their budget will be replenished. They are in the release-server list.

- **Ready-server list:** A priority queue containing all the active servers.

- **Release-server list:** A priority queue containing all the inactive servers. It keeps track of system event: replenishment of periodic servers.

- **Running server:** The only server from the ready-server list that is currently running. At every system tick, its remaining budget is decreased by one time unit, until it exhausts.

- **Idle server:** The lowest priority server that runs when no other server is active. In the deferrable server, it runs when there is no ready task in the system. This is useful for maintaining and testing the temporal separation among servers and also useful in testing system behavior. This information is useful in detecting over-reservations of server budgets and can be used as feedback to resource management.

- **Ready-task list:** Each subsystem maintains a separate ready-task list to keep track of its ready tasks. Only one ready-task list will be active at any time in the system: the ready list of the running server.

- **Idle task:** A lowest priority task existing in each server. It runs when its server has budget remaining but none of its task are ready to execute (in idling server). In deferrable server, the idle task of idle server will run instead.

6.4.6 Design Considerations

Here we present the challenges and goals of a HSF implementation that our implementation on FreeRTOS should satisfy:

1. **The use of HSF and the original FreeRTOS operating system:** User should be able to make a choice for using the HSF or the original FreeRTOS scheduler.

2. **Consistency with the FreeRTOS kernel and keeping its API intact:** To get minimal changes and better utilization of the system, it will be good to match the design of the HSF implementation with the underlying FreeRTOS operating system. This includes consistency from the naming conventions to API, data structures and the coding style. To increase the usability and understandability of HSF implementation for FreeRTOS users, major changes should not be made in the underlying kernel.

3. **Enforcement:** Enforcing server preemption at budget depletion; its currently executing task (if any) must be preempted and the server should be switched out. And similarly at budget replenishment, the server should become active; if its priority is highest among all the active servers then a server context-switch should be made and this server should execute.

4. **Monitoring budget consumption:** The budget consumption of servers should be monitored to properly handle server budget depletion (the tasks of the server should execute until its budget depletion).

5. **The temporal isolation among servers must be guaranteed:** When one server is overloaded and its task miss the deadlines, it must not affect the execution of other servers. Also when no task is active to consume its server's capacity; in the idling server this capacity should idle away while in deferrable server it should be preserved.

6. **Protecting against interference from inactive servers:** The inactive servers should not interfere in the execution of active servers.

7. **Minimizing the overhead of server context-switch and tick handler:** For an efficient implementation, design considerations should be made to reduce these overheads.

6.5 Implementation

The user needs to set a macro `configHIERARCHICAL_SCHEDULING` as `1` or `0` in the configuration file `FreeRTOSConfig.h` of the FreeRTOS to start the hierarchical scheduler or the original FreeRTOS scheduler. The server type can be set via macro `configGLOBAL_SERVER_MODE` in the configuration file, which can be idling periodic or deferrable server. We are using FPS with the FIFO (to break ties between equal priorities) at both levels. We have changed the FreeRTOS RR policy to FIFO for the local schedulers, in order to use HSF-analysis in future. Further RR is costly in terms of overhead (increased number of context switches).

Each server has a server control block, `subSCB`, containing the server's parameters and lists. The servers are created by calling the API `xServerCreate()` that creates an idling or deferrable server depending on the server type macro value, and do the server initializations which includes `subSCB` value's initialization, and initialization of server lists. It also creates an idle task in that server. An idle server with an idle task is also created to setup the system. The scheduler is started by calling `vTaskStartScheduler()` (typically at the end of the `main()` function), which is a non-returning function. Depending on the value of the `configHIERARCHICAL_SCHEDULING` macro, either the original FreeRTOS scheduler or the hierarchical scheduler will start execution. `vTaskStartScheduler()` then initializes the system-time to 0 by setting up the timer in hardware.

6.5.1 System Design

Here we describe the details of design, implementation, and functionality of the two-level HSF in FreeRTOS.

The Design of the Scheduling Hierarchy

The global scheduler maintains a running server pointer and two lists to schedule servers: a ready-server list and a release-server list. A server can be either in ready-server or release-server list at any time, and is implied as active or inactive respectively. Only one server from the ready-server list runs at a time.
Running server: The running server is identified by a pointer. This server has the highest priority among all the currently ready servers in the system.

At any time instance, only the tasks of the currently running server will run according to the fixed-priority scheduling policy. When a server context-switch

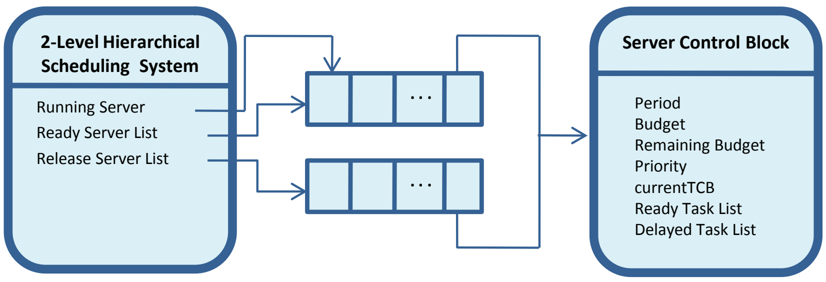


Figure 6.2: Data structures for active and inactive servers

occurs, the running server pointer is changed to the newly running server and all the tasks of the new running server become ready for execution.

Ready-server list: contains all the servers that are active (whose remaining budgets are greater than zero). This list is maintained as a double linked list. The `ListEnd` node contains two pointers; `ListEnd.previous` and `ListEnd.next` that point to the last node and first node of the list respectively as shown in Figure 6.3. It is the FreeRTOS structure of list, and provides a quick access to list elements, and very fast modifications of the list. It is ordered by the priority of servers, the highest priority ready server is the first node of the list.

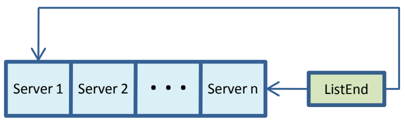


Figure 6.3: The structure of ready-server and release-server lists

Release-server list: contains all the inactive servers whose budget has depleted (their remaining budget is zero), and will be activated again at their next activation periods. This list is maintained as a double linked list as shown in Figure 6.3 and is ordered by the next replenishment time of servers, which is the absolute time when the server will become active again.

The Design of the Server

The local scheduler schedules the tasks that belong to a server in a fixed-priority scheduling manner. Each server is specified by a server Control Block, called `subSCB`, that contains all information needed by a server to run in the hierarchical scheduling, i.e. the period, budget, remaining budget, priority and the queues as presented in Figure 6.4.

Each server maintains a currently running task and two lists to schedule its tasks: a ready-task list, and a delayed-task list. Ready task and delayed task lists have the same structure as the FreeRTOS scheduler has. Delayed-task list is the FreeRTOS list and is used by the tasks that are delayed because of the FreeRTOS `vTaskDelay` or `vTaskDelayUntil` functions.

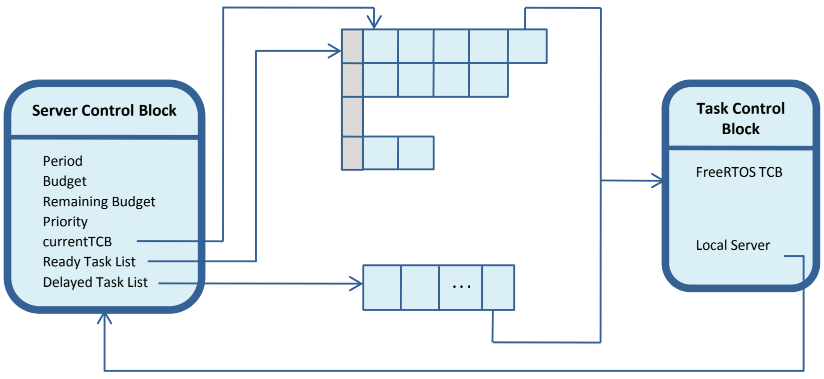


Figure 6.4: Data structures for ready and delayed tasks

Current running task: `currentTCB` is a FreeRTOS pointer that always points to the currently running task in the system. This is the task with the highest priority among all the currently ready tasks of the running server's ready task list.

Ready-task list: Each server maintains a separate ready-task list to keep track of its ready tasks. Only one ready-task list will be active at any time in the system: the ready list of the currently running server. When a server starts executing, its ready-task list becomes active, and `currentTCB` points to the highest priority task. This list is maintained in a similar way as FreeRTOS ready list, because we do not want to make major changes in the underlying operating system.

A separate ready-task list for each server reduces the server context-switch overhead, since the tasks swapping at every server context-switch is very costly. Further it also keeps our implementation consistent with the FreeRTOS.

The ready task list is an array of circular double linked lists of the tasks. The index of the array presents the priorities of tasks within a subsystem as shown by the gray color in Figure 6.5. By default, FreeRTOS uses 8 different priority levels for the tasks from lowest priority 1 (only idle task has priority 0) to the highest 7. (User can change it till the maximum 256 different task priorities). The tasks of the same priority are placed as a double linked list at the index of that particular priority. The last node of the double linked list at each index is End pointer that points to the previous (the last node of the list) and to the next (the first node of the list) as shown in the Figure 6.5.

For insertions the efficiency is $O(1)$, and for searching it is $O(n)$ in the worst case, where n is the maximum allowed priority for tasks in the subsystem.

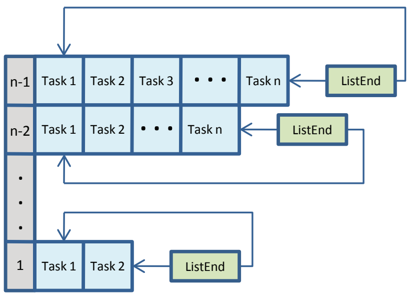


Figure 6.5: The structure of ready-task list

Tasks: We added a pointer to the task `TCB`, that points to its parent server control block to which this task belongs. This is the only addition done to the `TCB` of FreeRTOS to adopt it to the two-level hierarchical scheduling framework.

Server context-switch: Since each subsystem has its own ready list for its tasks, the server context switch is very light-weight. It is only the change of a pointer, i.e. from the task list of the currently executing server to the ready-task list of the newly running server. At this point, the ready-task list of the newly running server is activated and all the tasks of the list become ready for execution.

Task context-switch: We are using the FreeRTOS task context-switch which is very fast and efficient as evaluated in Section 6.6.2. At this point, the ready-task list of the newly running server is activated and all the tasks of the list become ready for execution.

6.5.2 System Functionality

The Functionality of the Tick Handler

The tick handler is executed at each system tick (1ms be default). At each tick interrupt:

- The system tick is incremented.
- Check for the server activation events. Here the activation time of (one or more) servers is checked and if it is equal to the system time then the server is replenished with its maximum budget and is moved to the ready-server list.
- The global scheduler is called to incorporate the server events.
- The local scheduler is called to incorporate the task events.

The Functionality of the Global Scheduler

In a two-level hierarchical scheduling system, a global scheduler schedules the servers (subsystems) in a similar fashion as the tasks are scheduled by a simple scheduler. The global scheduler is called by the `prvScheduleServers()` kernel function from within the tick-handler. The global scheduler performs the following functionality:

- At each tick interrupt, the global scheduler decrements the remaining budget B_s of the running server by one and handles budget expiration event (i.e. at the budget depletion, the server is moved from the ready-server list to the release-server list).
- Selects the highest priority ready server to run and makes a server context-switch if required. `prvChooseNextIdlingServer()` or `prvChooseNextDeferrableServer()` is called to select idling or deferrable server, depending on the `configGLOBAL_SERVER_MODE` macro. All the events that occurred during inactive state of the server (tasks activations) are handled here.
- `prvAdjustServerNextReadyTime(pxServer)` is called to set up the next activation time to activate the server periodically.

In idling server, `prvChooseNextIdlingServer()` simply selects the first node (with highest priority) from the ready-server list and makes it the current running server. While in case of deferrable server, the `prvChooseNextDeferrableServer()` function checks in the ready-server list for the next ready server that has any task ready to execute even if the currently running server has no ready task and its budget has not exhausted. It also handles the situation when the server's remaining budget is greater than 0, but its period ends, in this case the server is replenished with its full capacity.

The Functionality of the Local Scheduler

The local scheduler is called from within the tick interrupt using the adopted FreeRTOS kernel function `vTaskSwitchContext()`. The local scheduler is the original FreeRTOS scheduler with the following modifications:

- The round robin scheduling policy among equal priority tasks is changed to FIFO policy.
- Instead of a single ready-task or delayed-task list (as in original FreeRTOS), now the local scheduler accesses a separate ready-task and delayed-task list for each server.

6.5.3 Addressing Design Considerations

Here we address how we achieve the design requirements that are presented in Section 6.4.6.

1. **The use of HSF and the original FreeRTOS operating system:** We have kept all the original API of FreeRTOS, and the user can choose to run either the original FreeRTOS operating system or the HSF by just setting a macro `configHIERARCHICAL_SCHEDULING` to 0 or 1 respectively in the configuration file.
2. **Consistency with the FreeRTOS kernel and keeping its API intact:** We have kept consistency with the FreeRTOS from the naming conventions to the data structures used in our implementations; for example ready-task list, ready and release server lists. These lists are maintained in a similar way as of FreeRTOS. We have kept the original semantics of the API and the user can run the original FreeRTOS by setting `configHIERARCHICAL_SCHEDULING` macro to 0.

3. **Enforcement:** At each tick interrupt, the remaining budget of the running server is checked and at budget depletion (remaining budget becomes 0), the server is moved from active (ready-server list) to the inactive (release-server list) state. Moreover, release-server list is also checked for the periodic activation of servers at each system tick and at budget replenishment of any server, it is moved from inactive to active state. Preemptive scheduling policy makes it possible.

4. **Monitoring budget consumption:** The remaining budget variable of each server's `subSCB` is used to monitor the consumption. At each system tick, the remaining budget of the running server is decremented by one, and when it exhausts the server is moved from active to the inactive state.

5. **The temporal isolation among servers must be guaranteed:** We tested the system and an idle task runs when there is no task ready to execute. To test the temporal isolation among servers, we use an *Idle server* that runs when no other server is active. It is used in testing the temporal isolation among servers. Section 6.6.1 illustrates the temporal isolation.

6. **Protecting against interference from inactive servers:** The separation of active and inactive servers in separate server queues prevents the interference from inactive servers and also poses less overhead in handling system tick interrupts.

7. **Minimizing the overhead of server context-switch and tick handler:** A separate ready-task list for each subsystem reduces the task swapping overhead to only the change of a pointer. Therefore, the server context-switch is very light-weight. The access to such a structure of ready list is fast and efficient especially in both inserting and searching for elements. Further the tasks swapping at every server context-switch is very heavy in such a structure.

6.6 Experimental Evaluation

In this section, we present the evaluation of behavior and performance of our HSF implementation. All measurements are performed on the target platform EVK1100 [6]. The AVR32UC3A0512 micro-controller runs at the frequency of 12MHz and its tick interrupt handler at 1ms.

6.6.1 Behavior Testing

In this section we perform two experiments to test the behavior our implementation. Two servers S1, and S2 are used in the system, plus an idle server is created. The servers used to test the system are given in Table 6.1.

Server	S1	S2
Priority	2	1
Period	20	40
Budget	10	15

Table 6.1: Servers used to test system behavior.

Test1: This test is performed to check the behavior of idling periodic and deferrable servers by means of a trace of the execution. Task properties and their assignments to the servers is given in Table 6.2. Note that higher number means higher priority for both servers and tasks. The visualization of the execution for idling and deferrable servers is presented in Figure 6.6 and Figure 6.7 respectively.

Tasks	T1	T2	T3
Servers	$S1$	$S1$	$S2$
Priority	1	2	2
Period	20	15	60
Execution Time	4	2	10

Table 6.2: Tasks in both servers.

In the diagram, the horizontal axis represents the execution time starting from 0. In the task's visualization, the arrow represents task arrival, a gray rectangle means task execution, a solid white rectangle represents either local preemption by another task in the server or budget depletion, and a dashed white rectangle means the global preemption. In the server's visualization, the numbers along the vertical axis are the server's capacity, the diagonal line represents the server execution while the horizontal line represents either the waiting time for the next activation (when budget has depleted) or the waiting for its turn to execute (when some other server is executing).

The difference in idling and deferrable servers is clear from these Figures. In idling periodic servers, all the servers in the system executes till budget depletion, if no task is ready then the idle task of that server executes till its

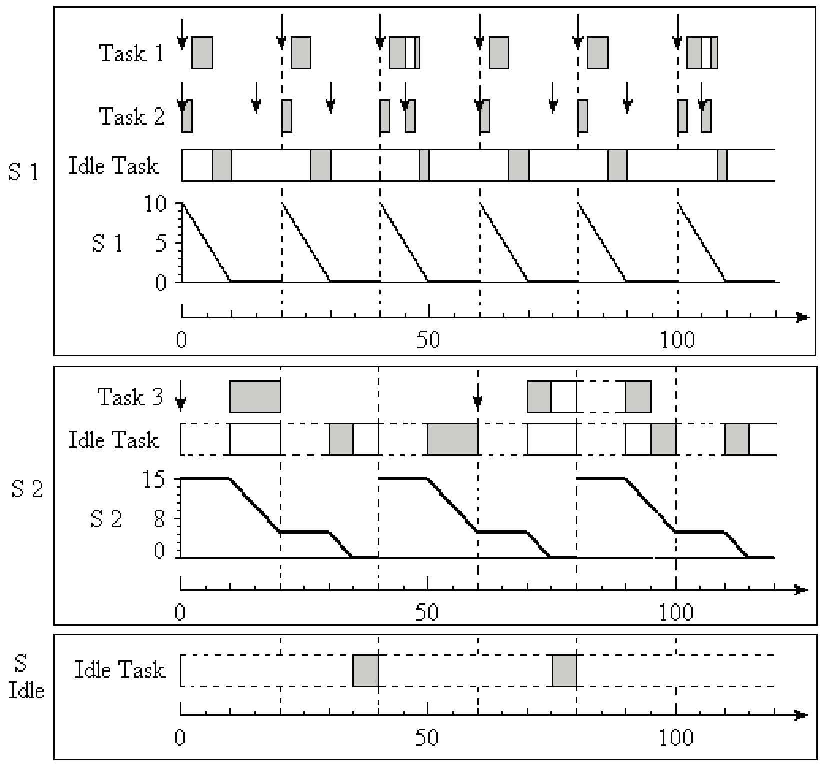


Figure 6.6: Trace for idling periodic servers

budget depletion. While in deferrable servers, when no task is ready in the server even if it has the capacity, the server will give the chance to another server to execute and preserves its capacity. Thats why there is no idle task (of S1 and S2) execution in deferrable servers as obvious from Figure 6.7. When no task is ready to execute in the system, then idle task of idle server will execute.

Test2: The purpose of this test is to evaluate the system behavior during the overload situation and to test the temporal isolation among the servers. For example, if one server is overloaded and its tasks miss deadlines, it must not affect the behavior of other servers in the system.

The same example is executed to perform this test but with the increased

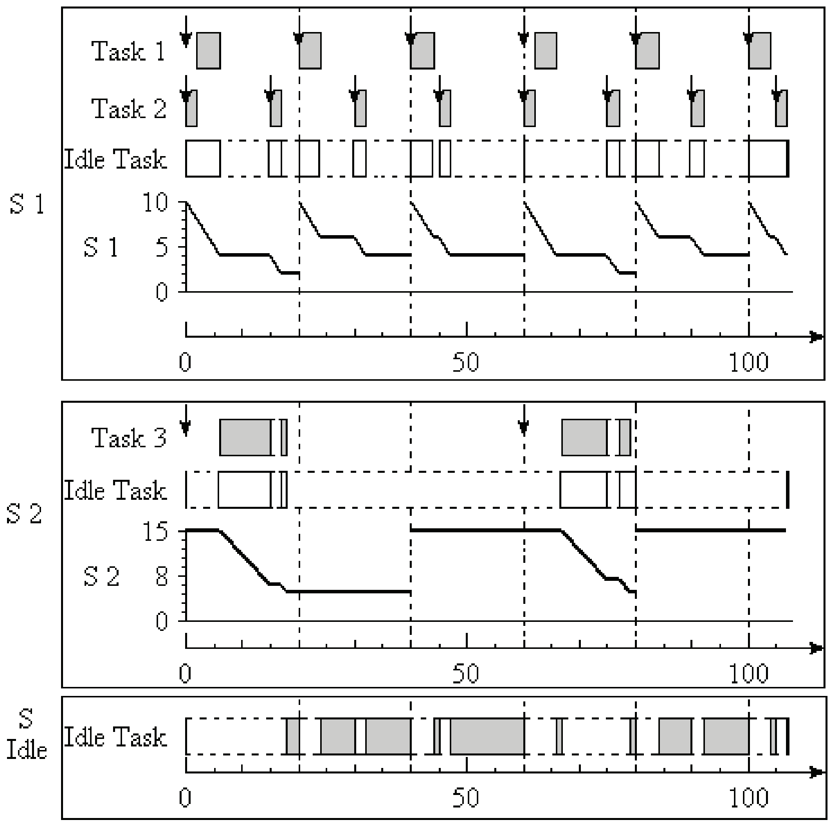


Figure 6.7: Trace for deferrable servers

utilization of S1. The execution times of T1 and T2 are increased to 4 and 6 respectively, hence making the server S1 utilization greater than 1. Therefore the low priority task T1 misses its deadlines as shown by solid black lines in the Figure 6.8. S1 is never idling because it is overloaded. It is obvious from Figure 6.8, that the overload of S1 does not effect the behavior of S2 even though it has low priority.

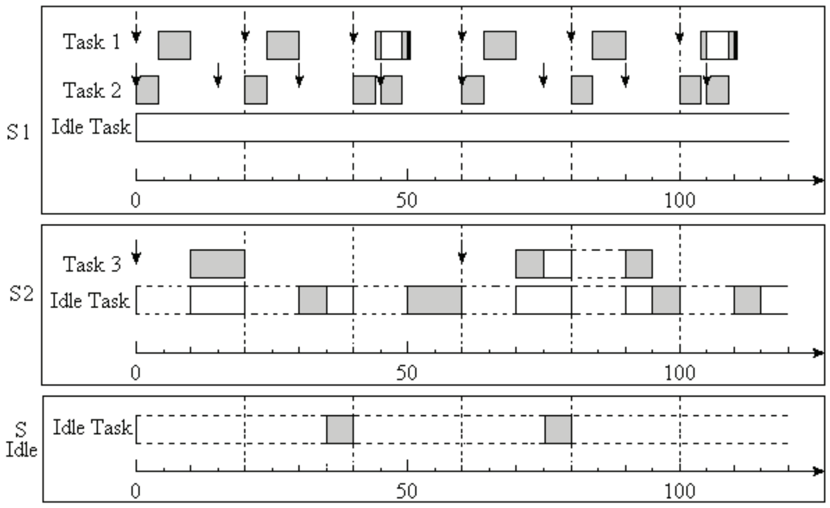


Figure 6.8: Trace showing temporal isolation among idling servers

6.6.2 Performance Assessments

Here we present the results of the overhead measurements for the idling and deferrable servers. The time required to run the global scheduler (to schedule the server) is the first extra functionality needed to be measured; it includes the overhead of server context-switch. The tick interrupt handler is the second function to be measured; it encapsulated the global scheduler within it, hence the overhead measurement for tick interrupt represents the sum of tick-increasing time and global scheduler time. The third overhead needed to be assessed is the task context-switch.

Two test scenarios are performed to evaluate the performance for both idling and deferrable servers. For each measure, a total of 1000 values are computed. The minimum, maximum, average and standard deviation on these values are calculated and presented for both types of servers. All the values are given in micro-seconds (μs).

Test Scenario 1

For the first performance test, 3 servers, S1, S2, and S3 are created with a total of 7 tasks. S1 contains 3 tasks while S2 and S3 has 2 tasks each. The measurements are extracted for task and server context-switches, global scheduler and tick interrupt handler and are reported below.

Task context switch: The FreeRTOS context-switch is used for doing task-level switching. We found it very efficient, consistent and light-weight, i.e. 10μs always as obvious from Table 6.3.

Server type	Min.	Max.	Average	St. Deviation
`Idling`	10	10	10	0
`Deferrable`	10	10	10	0

Table 6.3: The task context-switch measures for both servers.

Choosing next server: It is *fetching the highest priority server (first node from the server ready queue)*, and it is very fast for both types of servers as given in Table 6.4. Note that the situations where there is no need to change the server, it becomes 0 and this situation is excluded from these results.

Server type	Min.	Max.	Average	St. Deviation
`Idling`	10	10	10	0
`Deferrable`	10	32	14.06593	5.6222458

Table 6.4: The server context-switch measures for both servers.

The deferrable overhead is greater than idling server because of the increased functionality, as explained in Section 6.5.2.

Global scheduler: The WCET of the global scheduler is dependent on the number of events it handles. As explained in Section 6.5.2, the global scheduler handles the server activation events and the events which has been postponed during inactive time in this server, therefore, its execution time depends on the number of events. The overhead measures for global scheduler function to execute for both types of servers are given in Table 6.5.

Server type	Min.	Max.	Average	St. Deviation
`Idling`	10	53	12.33666	6.0853549
`Deferrable`	10	42	13.34865	7.5724052

Table 6.5: The global scheduler overhead measures for both servers.

Tick interrupt handler: It includes the functionality of global and local schedulers. The WCET of the tick handler is dependent on the number of servers and tasks in the system. Note that the task context-switch time is excluded from this measurement.

Server type	Min.	Max.	Average	St. Deviation
`Idling`	32	74	37.96903	7.00257381
`Deferrable`	32	85	41.17582	10.9624383

Table 6.6: The tick interrupt overhead measures for both servers.

Again the deferrable overhead is greater than that of the idling server because of the increased functionality and increased number of server context-switches at run-time.

Test Scenario 2

The experiments are run to check heavy system loads. The setup includes $10, 20, 30$, and 40 servers in the system, each running a single task in it. We cannot create more than 40 idling servers, and more than 30 deferrable servers due to memory limitations on our hardware platform. For this test scenario we only measured the overheads for the global scheduler and the tick interrupt handler, because choosing next server is part of global scheduler and because the time to execute task context-switch is not affected by the increase of number of servers in the system.
Global scheduler: The values for idling and deferrable servers are presented in Table 6.7 and 6.8 respectively.

Number of servers	Min.	Max.	Average	St. Deviation
10	10	21	10.0439	0.694309682
20	10	32	10.1538	1.467756006
30	10	32	10.3956	2.572807933
40	10	32	10.3186	2.258614766

Table 6.7: The global scheduler overhead measures for idling server.

The global scheduler's overhead measures are dependent on the number of events it handles as explained in Section 6.5.2. In this test scenario, there is only one task per server, that reduces the number of events to be handled by the global scheduler, therefore, the maximum overhead values in Table 6.7 are less than from those of Table 6.5. The same reasoning stands for deferrable server too.

Number of servers	Min.	Max.	Average	St. Deviation
10	10	53	25.84	8.950729331
20	10	53	25.8434	11.90195638
30	10	53	27.15	9.956851354

Table 6.8: The global scheduler overhead measures for deferrable server.

Tick interrupt handler: The measured overheads for idling and deferrable servers are reported in Table 6.9 and 6.10 respectively. These do not include the task context-switch time.

Number of servers	Min.	Max.	Average	St. Deviation
10	53	96	64.57742	4.656420272
20	96	106	98.35764	4.246876974
30	128	138	132.2058	4.938988398
40	160	181	164.8022	5.986888605

Table 6.9: The tick interrupt overhead measures for idling servers.

From Tables 6.6, 6.9, 6.10 it is clear that the tick interrupt overhead increases with the increase in the number of servers in the system.

Number of servers	Min.	Max.	Average	St. Deviation
10	106	128	126.2574	4.325860528
20	140	149	144.5446	4.522222357
30	172	181	178.7723	3.903539901

Table 6.10: The tick interrupt overhead measures for deferrable servers.

6.6.3 Summary of Evaluation

We have evaluated our implementation on an actual real environment i.e. a 32-bit EVK1100 board hence our results are more valid than simulated results like [17] where the simulation experiments are simulated for OpenRisc 1000 architecture and hence having a very precise environmental behavior. We have evaluated the behavior and performance of our implementation for *resource allocation during heavy load,* and *overload* situations, and found that it behaves correctly and gives very consistent results.

We have also evaluated the efficiency of our implementation, i.e. the efficiency of task context-switch, global scheduler, and tick handler. Searching for the highest priority server and task the efficiencies are $O(1)$ and $O(n)$ respectively, where n is the maximum allowed priority for tasks in the subsystem. For insertions, it is $O(m)$ and $O(1)$ for the server and task respectively, where m is the number of servers in the system in the worst case. Our results for task context-switch, and choosing scheduler conforms this efficiency as compared to [17], where the efficiency is also dependent on dummy events in the RELTEQ queues. These dummy events are not related to the scheduler or tasks, but to the RELTEQ queue management.

6.7 Conclusions

In this paper, we have implemented a two-level hierarchical scheduling support in an open source real-time operating system, FreeRTOS, to support temporal isolation among real-time components. We have implemented idling periodic and deferrable servers using fixed-priority preemptive scheduling at both local and global scheduling levels. We focused on being consistent with the underlying operating system and doing minimal changes to get better utilization of the system. We presented our design details of two-level HSF and kept the original FreeRTOS API semantics.

We have tested our implementations and presented our experimental evaluations performed on EVK1100 AVR32UC3A0512 micro-controller. We have checked it during heavy-load and over-load situations and have reported our results. It is obvious from the results of the overhead measurements (of tick handler, global scheduler, and task context-switch) that the design decisions made and the implementation is very efficient.

In the future we plan to implement support for legacy code in our HSF implementation for the FreeRTOS i.e. to map the FreeRTOS API to the new API, so that the user can run her/his old code in a subsystem within the HSF. We will implement the periodic task model and a lock-based synchronization protocol [25] for global resource sharing among servers. We also want to improve the current Priority Inheritance Protocol for local resource sharing of FreeRTOS by implementing Stack Resource Protocol. And finally we want to integrate this work within the virtual node concept [5].

6.8 Appendix

A synopsis of the application program interface of HSF implementation is presented below. The names of these API and macros are self-explanatory.
The newly added user API and macro are the following:

1. `signed portBASE_TYPE xServerCreate(xPeriod, xBudget, uxPriority, *pxCreatedServer);`
2. `signed portBASE_TYPE xServerTaskGenericCreate( pxTaskCode, pcName, usStackDepth, *pvParameters, uxPriority, *pxCreatedTask, pxCreatedServer, *puxStackBuffer, xRegions ) PRIVILEGED_FUNCTION;`
3. `#define xServerTaskCreate( pvTaskCode, pcName, usStackDepth, pvParameters, uxPriority, pxCreatedTask, pxCreatedServer ) xServerTaskGenericCreate( (pvTaskCode), (pcName), (usStackDepth), (pvParameters), (uxPriority), (pxCreatedTask), (pxCreatedServer), ( NULL ), ( NULL ))`
4. `portTickType xServerGetRemainingBudget( void );`

The newly added private functions and macros are as follows:

1. `#define prvAddServerToReadyQueue( pxSCB )`
2. `#define prvAddServerToReleaseQueue( pxSCB )`
3. `#define prvAddServerToOverflowReleaseQueue( pxSCB )`
4. `#define prvChooseNextDeferrableServer( void )`
5. `#define prvChooseNextIdlingServer( void )`
6. `static inline void prvAdjustServerNextReadyTime( *pxServer );`
7. `static void prvInitialiseServerTaskLists( *pxServer );`
8. `static void prvInitialiseGlobalLists(void);`
9. `static signed portBASE_TYPE prxRegisterTasktoServer(* pxNewTCB, *pxServer);`
10. `static signed portBASE_TYPE prxServerInit(* pxNewSCB);`
11. `static signed portBASE_TYPE xIdleServerCreate(void);`
12. `static void prvScheduleServers(void);`
13. `static void prvSwitchServersOverflowDelayQueue(* pxServerList);`
14. `static void prvCheckServersDelayQueue(* pxServerList);`

We adopted the following user APIs to incorporate HSF implementation. The original semantics of these API is kept and used when the user run the original FreeRTOS by setting `configHIERARCHICAL_SCHEDULING` macro to 0.

1. `signed portBASE_TYPE xTaskGenericCreate( pxTaskCode, pcName, usStackDepth, *pvParameters, uxPriority, *pxCreatedTask, *puxStackBuffer, xRegions );`
2. `void vTaskStartScheduler( void );`
3. `void vTaskStartScheduler (void);`
4. `void vTaskDelay( xTicksToDelay );`
5. `void vTaskDelayUntil( pxPreviousWakeTime, xTimeIncrement);`

and adopted private functions and macros:

1. `#define prvCheckDelayedTasks(pxServer)`
2. `#define prvAddTaskToReadyQueue( pxTCB )`
3. `void vTaskIncrementTick( void );`
4. `void vTaskSwitchContext( void );`

Bibliography

[1] L. Sha, T. Abdelzaher, K-E. rzn, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real Time Scheduling Theory: A Historical Perspective. *Real-Time Systems*, 28(2/3):101–155, 2004.

[2] J.A. Stankovic, M. Spuri, M. Di Natale, and G.C. Buttazzo. Implications of Classical Scheduling Results for Real-Time Systems. *IEEE Computer*, pages 16–25, June 1995.

[3] Z. Deng and J. W.-S. Liu. Scheduling real-time applications in an open environment. In *Proc. 18th IEEE Real-Time Systems Symposium (RTSS)*, 1997.

[4] FreeRTOS web-site. http://www.freertos.org/.

[5] Rafia Inam, Jukka Mäki-Turja, Jan Carlson, and Mikael Sjödin. Using temporal isolation to achieve predictable integration of real-time components. In *22nd Euromicro Conference on Real-Time Systems (ECRTS10) WiP Session*, pages 17–20, July 2010.

[6] ATMEL EVK1100 product page. http://www.atmel.com/dyn/Products/tools_card.asp?tool_id=4114.

[7] Daeyoung Kim, Yann-Hang Lee, and M. Younis. Spirit-ukernel for strongly partitione real-time systems. In *Proc. of the 7th International conference on Real-Time Computing Systems and Applications (RTCSA'00)*, 2000.

[8] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proc. 24th IEEE Real-Time Systems Symposium (RTSS)*, pages 2–13, 2003.

[9] T.-W. Kuo and C.-H. Li. A fixed-priority-driven open environment for real-time applications. In *Proc. 20th IEEE Real-Time Systems Symposium (RTSS)*, 1999.

[10] G. Lipari and S.Baruah. Efficient scheduling of real-time multi-task applications in dynamic systems. In *Proc. 6th IEEE Real-Time Technology and Applications Symposium (RTAS)*, pages 166–175, 2000.

[11] L. Almeida and P. Pedreiras. Scheduling within temporal partitions: response-time analysis and server design. In *ACM Intl. Conference on Embedded Software(EMSOFT'04)*, pages 95–103, 2004.

[12] X. Feng and A. Mok. A model of hierarchical real-time virtual resources. In *IEEE Real-Time Systems Symposium (RTSS'02)*, pages 26–35, 2002.

[13] R. I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *IEEE Real-Time Systems Symposium (RTSS'05)*, pages 389–398, 2005.

[14] S. Saewong and R. Rajkumar. Hierarchical reservation support in resource kernels. In *Proc. 22th IEEE Real-Time Systems Symposium (RTSS)*, 2001.

[15] G. Buttazzo and P. Gai. Efficient edf implementation for small embedded systems. In *International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'06)*, 2006.

[16] Moris Behnam, Thomas Nolte, Insik Shin, Mikael Åsberg, and Reinder J. Bril. Towards hierarchical scheduling on top of vxworks. In *Proceedings of the Fourth International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'08)*, pages 63–72, July 2008.

[17] Mike Holenderski, Wim Cools, Reinder J. Bril, and J. J. Lukkien. Extending an Open-source Real-time Operating System with Hierarchical Scheduling. Technical Report, Eindhoven University, 2010.

[18] Mike Holenderski, Wim Cools, Reinder J. Bril, and J. J. Lukkien. Multiplexing Real-time Timed Events. In *Work in Progress session of the IEEE International Conference on Emerging Techonologies and Factory Automation (ETFA09)*, 2009.

[19] M.M.H.P. van den Heuvel, Mike Holenderski, Wim Cools, Reinder J. Bril, and Johan J. Lukkien. Virtual Timers in Hierarchical Real-time Systems. In *Work in Progress Session of the IEEE Real-Time Systems Symposium (RTSS09)*, December 2009.

[20] Håkan Sundell and Philippas Tsigas. Simple Wait-Free Snapshots for Real-Time Systems with Sporadic Tasks. In *Proceedings of the (RTCSA 2004)*, pages 325–240.

[21] Microchip web-site.

[22] EE TIMES web-site. http://www.eetimes.com/design/embedded/4008920/The-results-for-2010-are-in-.

[23] L. Sha, J.P. Lehoczky, and R. Rajkumar. Solutions for some Practical problems in Prioritised Preemptive Scheduling. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, pages 181–191, 1986.

[24] J.K. Strosnider, J.P. Lehoczky, and L. Sha. The deferrable server algorithm for Enhanced Aperiodic Responsiveness in Hard Real-time Environments. *IEEE Transactions on Computers*, 44(1), 1995.

[25] Moris Behnam, Thomas Nolte, Mikael Sjödin, and Insik Shin. Overrun Methods and Resource Holding Times for Hierarchical Scheduling of Semi-Independent Real-Time Systems. *IEEE Transactions on Industrial Informatics*, 6(1), February 2010.

Chapter 7

Paper C: Hard Real-time Support for Hierarchical Scheduling in FreeRTOS

Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, Moris Behnam
In Proceedings of the 7^{th} International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT' 11), Pages 51-60, Porto, Portugal, July, 2011.

Abstract

This paper presents extensions to the previous implementation of two-level Hierarchical Scheduling Framework (HSF) for FreeRTOS. The results presented here allow the use of HSF for FreeRTOS in hard-real time applications, with the possibility to include legacy applications and components not explicitly developed for hard real-time or the HSF.

Specifically, we present the implementations of (i) global and local resource sharing using the Hierarchical Stack Resource Policy and Stack Resource Policy respectively, (ii) kernel support for the periodic task model, and (iii) mapping of original FreeRTOS API to the extended FreeRTOS HSF API. We also present evaluations of overheads and behavior for different alternative implementations of HSRP with overrun from experiments on the AVR 32-bit board EVK1100. In addition, real-time scheduling analysis with models of the overheads of our implementation is presented.

7.1 Introduction

In real-time embedded systems the components and components integration must satisfy both (1) functional correctness and (2) extra-functional correctness, such as satisfying timing properties. Hierarchical Scheduling Framework (HSF) [1] has emerged as a promising technique in satisfying timing properties while integrating complex real-time components on a single node. It supplies an effective mechanism to provide temporal partitioning among components and supports independent development and analysis of real-time systems [2]. In HSF, the CPU is partitioned into a number of subsystems (servers or applications); each real-time component is mapped to a subsystem that contains a local scheduler to schedule the internal tasks of the subsystem. Each subsystem performes its own task scheduling, and the subsystems are scheduled by a global (system-level) scheduler. Two different synchronization mechanisms *overrun* [3] and *skipping* [4] have been proposed and analyzed for inter-subsystem resource sharing, but not much work has been performed for their practical implementations.

We have chosen FreeRTOS [5], a portable open source real-time scheduler to implement hierarchical scheduling framework. The goal is to use the HSF-enabled FreeRTOS to implement the *virtual node* concept in the ProCom component-model [6, 7]. FreeRTOS has been chosen due to its main features, like it's open source nature, small size and scalability, and support of many different hardware architectures allowing it to be easily extended and maintained. Our HSF implementation [8] on FreeRTOS for idling periodic and deferrable servers uses fixed priority preemptive scheduling (FPPS) for both global and local-level scheduling. FPPS is flexible and simple to implement, plus is the de-facto industrial standard for task scheduling. In this paper we extend our implementation of HSF to support hard real-time components. We implement time-triggered periodic tasks within the FreeRTOS operating system. We improve the resource sharing policy of FreeRTOS, and implement support for inter-subsystem resource sharing for our HSF implementation. We also provide legacy support for existing systems or components to be executed within our HSF implementation as a subsystem.

7.1.1 Contributions

The main contributions of this paper are:

- We have supported *periodic task model* within the FreeRTOS operating system.

- We have provided *a legacy support* in our HSF implementation and have mapped the old FreeRTOS API to the new API so that the user can very easily use an old system into a server within a two-level HSF.
- We have provided an efficient implementation for *resource sharing* for our HSF implementation. This entails: support for *Stack Resource Policy* for local resource sharing, and *Hierarchical Stack Resource Policy* for global resource sharing with three diferent methods to handle *overrun*.
- We have included the *runtime overhead* for *local and global schedulability analysis* of our implementation.
- We describe the *detailed design* of all the above mentioned improvements in our HSF implementations with the consideration of minimal modifications in underlying FreeRTOS kernel.
- And finally, we have *tested and calculated the performance measures* for our implementations on an AVR-based 32-bit board EVK1100 [9].

7.1.2 Resource Sharing in Hierarchical Scheduling Framework

A two-level HSF [10] can be viewed as a tree with one parent node (global scheduler) and many leaf nodes (local schedulers) as illustrated in Figure 7.1. The leaf nodes contain its own internal set of tasks that are scheduled by a local (subsystem-level) scheduler. The parent node is a global scheduler and is responsible for dispatching the subsystems according to their resource reservations. Using HSF, subsystems can be developed and analyzed in isolation from each other.

In a two-level HSF the resources can be shared among tasks of the same subsystem (or intra-subsystem), normally referred as *local shared resource*. The resources can also be shared among tasks of different subsystems (or intersubsystem) called *global shared resources* as shown in Figure 7.1.

Different synchronization protocols are required to share resources at local and global levels, for example, *Stack Resource Policy (SRP)* [11] can be used at local level with FPPS, and to implement SRP-based overrun mechanism at global level, *Hierarchical Stack Resource Policy (HSRP)* [3] can be used.

Organisation: Section 7.2 presents the related work on hierarchical scheduler implementations. Section 7.3 gives a background on FreeRTOS in 7.3.1, a review of our HSF implementation in FreeRTOS in 7.3.2, and resource sharing techniques in HSF in section 7.3.3. In section 7.4 we provide our system

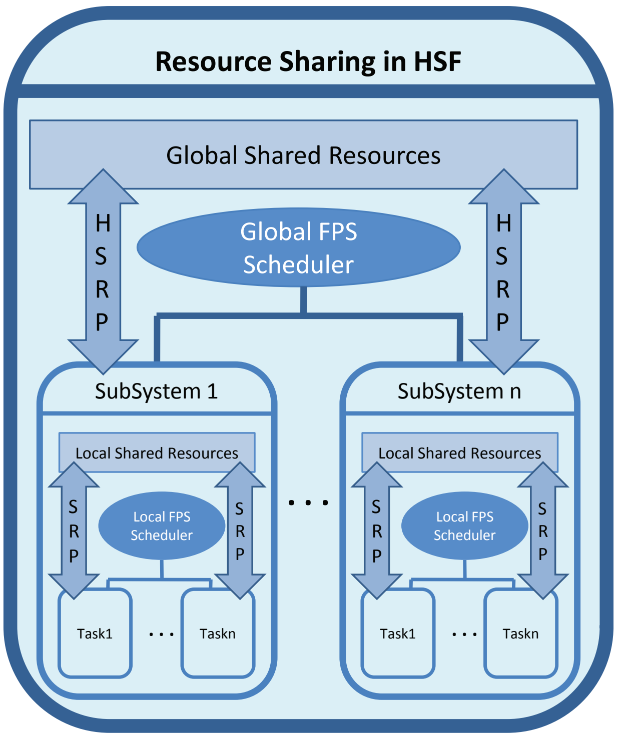


Figure 7.1: Two-level Hierarchical Scheduling Framework

model. We explain the implementation details of periodic task model, legacy support, and resource sharing in section 7.5. In section 7.6 we provide scheduling analysis and in section 7.7 we present the behavior of implementation and some performance measures. In section 7.8 we conclude the paper. The API for the local and the global resource sharing in HSF is given in Appendix.

7.2 Related Work

HSF has attained a substantial importance since introduced in 1990 by Deng and Liu [1]. Saewong and Rajkumar [12] implemented and analyzed HSF in CMU's Linux/RK with deferrable and sporadic servers using hierarchical deadline monotonic scheduling. Buttazzo and Gai [13] present an HSF implementation based on Implicit Circular Timer Overflow Handler (ICTOH) using EDF scheduling for an open source RTOS, ERIKA Enterprise kernel. A micro kernel called SPIRIT-μKernel is proposed by Kim *et al.* [10] based on two-level hierarchical scheduling methodology and demonstrate the concept, by porting two different application level RTOS, VxWorks and eCos, on top of the SPIRIT-μKernel. It uses an offline scheduler at global level and the fixed-priority scheduling at local level to schedule the partitions and tasks respectively. A detailed related work on HSF implementation without resource sharing is presented in [8].

7.2.1 Local and Global Synchronization Protocols

Local synchronization protocols

Priority inheritance protocol (PIP) [14] was developed to solve the priority inversion problem but it does not solve the chained blocking and deadlock problems. Sha *et al.* proposed the priority ceiling protocol (PCP) [14] to solve these problems. A slightly different alternative to PCP is the immediate inheritance protocol (IIP). Baker presented the stack resource policy (SRP) [11] that supports dynamic priority scheduling policies. For fixed-priority scheduling, SRP has the same behavior as IIP. SRP reduces the number of context-switches and the resource holding time as compared to PCP. Like most real-time operating systems, FreeRTOS only support an FPPS scheduler with PIP protocol for resource sharing. We provide support for SRP for local-level resource sharing in HSF.

Global synchronization protocols

For global resource sharing some additional protocols have been proposed. Fisher *et al.* proposed Bounded delay Resource Open Environment (BROE) [15] protocol for global resource sharing under EDF scheduling. Hierarchical Stack Resource Policy (HSRP) [3] uses the overrun mechanism to deal with the subsystem budget expiration within the critical section and uses two mechanisms (with pay back and without payback) to deal with the overrun. Subsystem Integration and Resource Allocation Policy (SIRAP) [4] uses the skipping mechanism to avoid the problem of subsystem budget expiration within the critical section. Both HSRP and SIRAP assume FPPS. The original HSRP [3] does not support the independent subsystem development for its analysis.

Behnam *et al.* [16] not only extended the analysis for the independent subsystem development, but also proposed a third form of overrun mechanism called extended overrun. In this paper we use HSRP for global resource sharing and implement all the three forms of the overrun protocol.

7.2.2 Implementations of Resource Sharing in HSF

Behnam *et al.* [17] present an implementation of a two-level HSF in the commercial operating system VxWorks with the emphasis of not modifying the underlying kernel. The implementation supports both FPS and EDF at both global and local level of scheduling and a one-shot timer is used to trigger schedulers. In [18], they implemented overrun and skipping techniques at the top of their FPS HSF implementation and compared the two techniques.

Holenderski *et al.* implemented a two-level fixed-priority HSF in μC/OS-II, a commercial real-time operating system [19]. This implementation is based on Relative Timed Event Queues (RELTEQ) [20] and virtual timers [21] on the top of RELTEQ to trigger timed events. They incorporated RELTEQ queues and virtual timers within the operating system kernel and provided interfaces for it and HSF implementation uses these interfaces. More recently, they extended the HSF with resource sharing support [22] by implementing SIRAP and HSRP (with and without payback). They measured and compared the system overheads of both primitives.

The work presented in this paper is different from that of [18] in the sense that we implement resource sharing in a two-level HSF with the aim of simplified implementation while adopting the kernel with the consideration of being consistent with the FreeRTOS. The user should be able to choose the original FreeRTOS or HSF implementation to execute, and also able to run legacy code within HSF with doing minimal changes in it. The work of this paper is different from that of [22] in the sense that we only extend the functionality of the operating system by providing support for HSF, and not changing or modifying the internal data structures. It aims at simplified implementation while minimizing the modifications of the underlying operating system. Our implementation is simpler than both [18, 22] since we strictly follow the rules of HSRP [3]. We do not have local ceilings for the global shared resources (as in [18, 22]) which simplifies the implementation. We do not allow local preemptions while holding the global resources which reduces the resource holding times as compared to [18, 22]. Another difference is that both [18, 22] implemented SIRAP and HSRP (with and without payback) while we implement all the three forms of overrun (with payback, without payback, and enhanced overrun). We do not support SIRAP because it is more difficult to use; the application programmer needs to know the WCET of each critical section to use SIRAP. Further neither implementation does provide analysis for their implementations.

7.3 Background

7.3.1 FreeRTOS

FreeRTOS is a portable, open source (licensed under a modified GPL), mini real-time operating system developed by Real Time Engineers Ltd. It is ported to 23 hardware architectures ranging from 8-bit to 32-bit micro-controllers, and supports many development tools. Its main advantages are portability, scalability and simplicity. The core kernel is simple and small, consisting of three or four (depends on the usage of coroutines) C files with a few assembler functions, with a binary image between 4 to 9KB.

Since most of the source code is in C language, it is readable, portable, and easily expandable and maintainable. Features like ease of use and understandability makes it very popular. More than $77,500$ official downloads in 2009 [23], and the survey result performed by professional engineers in 2010 puts the FreeRTOS at the top for the question "which kernel are you considering using this year" [24] showing its increasing popularity.

FreeRTOS kernel supports preemptive, cooperative, and hybrid scheduling. In the fixed-priority preemptive scheduling, the tasks with the same priority are scheduled using the round-robin policy. It supports both tasks and subroutines; the tasks with maximum 256 different priorities, any number of tasks and very efficient context switch. FreeRTOS supports both static and dynamic (changed at run-time) priorities of the tasks. It has semaphores and mutexes for resource sharing and synchronization, and queues for message passing among tasks. Its scheduler runs at the rate of one tick per milli-second by default.

FreeRTOS Synchronization Protocol: FreeRTOS supports basic synchronization primitives like *binary, counting* and *recursive semaphore*, and *mutexes*. The mutexes employ *priority inheritance protocol*, that means that when a higher priority task attempts to obtain a mutex that is already blocked by a lower priority task, then the lower priority task temporarily inherits the priority of higher priority task. After returning the mutex, the task's priority is lowered back to its original priority. Priority inheritance mechanism minimizes the *priority inversion* but it cannot cure deadlock.

7.3.2 A Review of HSF Implementation in FreeRTOS

A brief overview of our two-level hierarchical scheduling framework implementation [8] in FreeRTOS is given here.

Both global and local schedulers support fixed-priority preemptive scheduling (FPPS). Each subsystem is executed by a server S_s, which is specified by a *timing interface* $S_s(P_s, Q_s)$, where P_s is the period for that server $(P_s > 0)$, and Q_s is the capacity allocated periodically to the server $(0 < Q_s \leq P_s)$. Each server has a unique priority p_s and a remaining budget during the runtime of subsystem B_s. Since the previous implementation not focus on real-time, we only characterize each task τ_i by its priority ρ_i.

The global scheduler maintains a pointer, running server, that points to the currently running server.

The system maintains two priority-based lists. First is the ready-server list that contains all the servers that are ready (their remaining budgets are greater than zero) and is arranged according to the server's priority, and second is the release-server list that contains all the inactive servers whose budget has depleted (their remaining budget is zero), and will be activated again at their next activation periods and is arranged according to the server's activation times.

Each server within the system also maintains two lists. First is the ready-task list that keeps track of all the ready tasks of that server, only the ready list of the currently running server will be active at any time, and second is the delayed-task list of FreeRTOS that is used to maintain the tasks when they are not ready and waiting for their activation.

The hierarchical scheduler starts by calling `vTaskStartScheduler()` API and the tasks of the highest priority ready server starts execution. At each tick interrupt,

- The system tick is incremented.
- Check for the server activation events. The newly activated server is replenished with its maximum budget and is moved to the ready-server list.
- The global scheduler is called to handle the server events.
- The local scheduler is called to handle the task events.

The Functionality of the Global Scheduler

The global scheduler performs the following functionality:

- At each tick interrupt, the global scheduler decrements the remaining budget B_s of the running server by one and handles budget expiration event (i.e. at the budget depletion, the server is moved from the ready-server list to the release-server list).
- Selects the highest priority ready server to run and makes a server context-switch if required. Either `prvChooseNextIdlingServer()` or `prvChooseNextDeferrableServer()` is called to select idling or deferrable server, depending on the value of the `configGLOBAL_SERVER_MODE` macro in the `FreeRTOSConfig.h` file.
- `prvAdjustServerNextReadyTime(pxServer)` is called to set up the next activation time to activate the server periodically.

In idling server, the `prvChooseNextIdlingServer()` function selects the first node (with highest priority) from the ready-server list and makes it the current running server. While in case of deferrable server, the `prvChooseNextDeferrableServer()` function checks in the ready-server list for the next ready server that has any task ready to execute when the currently running server has no ready task even if it's budget is not exhausted. It also handles the situation when the server's remaining budget is greater than 0, but its period ends, in this case the server is replenished with its full capacity.

The Functionality of the Local Scheduler

The local scheduler is called from within the tick interrupt using the adopted FreeRTOS kernel function `vTaskSwitchContext()`. The local scheduler is the original FreeRTOS scheduler with the following modifications:

- The round robin scheduling policy among equal priority tasks is changed to FIFO policy to reduce the number of task context-switches.
- Instead of a single ready-task or delayed-task list (as in original FreeRTOS), now the local scheduler accesses a separate ready-task and delayed-task list for each server.

7.3.3 Resource Sharing in HSF

Stack Resource Policy at global and local levels: We have implemented the HSRP [3] which extends SRP to HSRP. The SRP terms are extended as follows:

- *Priority.* Each task has a priority ρ_i. Similarly, each subsystem has an associated priority p_s.
- *Resource ceiling.* Each globally shared resource R_j is associated with a resource ceiling for global scheduling. This global ceiling is the highest priority of any subsystem whose task is accessing the global resource. Similarly each locally shared resource also has a resource ceiling for local scheduling. This local ceiling is the highest priority of any task (within the subsystem) using the resource.
- *System/subsystem ceilings.* System/subsystem ceilings are dynamic parameters that change during runtime. The system/subsystem ceiling is equal to the currently locked highest global/local resource ceiling in the system/subsystem.

Following the rules of SRP, a task τ_i can preempt the currently executing task within a subsystem only if τ_i has a priority higher than that of running task and, at the same time, the priority of τ_i is greater than the current subsystem ceiling.

Following the rules of HSRP, a task τ_i of the subsystem S_i can preempt the currently executing task of another subsystem S_j only if S_i has a priority higher than that of S_j and, at the same time, the priority of S_i is greater than the current system ceiling. Moreover, whilst a task τ_i of the subsystem S_i is accessing a global resource, no other task of the same subsystem can preempt τ_i.

7.3.4 Overrun Mechanisms

This section explains three overrun mechanisms that can be used to handle budget expiry during a critical section in the HSF. Consider a global scheduler that schedules subsystems according to their periodic interfaces . The subsystem budget Q_s is said to expire at the point when one or more internal tasks have executed a total of Q_s time units within the subsystem period P_s. Once the budget is expired, no new task within the same subsystem can initiate its execution until the subsystems budget is replenished at the start of next subsystem period.

To prevent excessive priority inversion due to global resource lock its desirable to prevent subsystem rescheduling during critical sections of global resources. In this paper, we employ the overrun strategy to prevent such rescheduling. Using overrun, when the budget of subsystem expires and it has a task that is still locking a global shared resource, the task continues its execution until it releases the resource. The extra time needed to execute after the budget expiration is denoted as *overrun time* θ. We implement three different overrun mechanisms [16]:

1. The basic overrun mechanism without payback, denoted as BO: here no further actions will be taken after the event of an overrun.

2. The overrun mechanism with payback, denoted as PO: whenever overrun happens, the subsystem S_s pays back in its next execution instant, i.e., the subsystem budget Q_s will be decreased by θ_s i.e. $(Q_s - \theta_s)$ for the subsystems execution instant following the overrun (note that only the instant following the overrun is affected even if $\theta_s > Q_s$).

3. The enhanced overrun mechanism with payback, denoted as EO: It is based on imposing an offset (delaying the budget replenishment of subsystem) equal to the amount of the overrun θ_s to the execution instant that follows a subsystem overrun, at this instant, the subsystem budget is replenished with $Q_s - \theta_s$.

7.4 System Model

In this paper, we consider a two-level hierarchical scheduling framework, in which a global scheduler schedules a system S that consists of a set of independently developed and analyzed subsystems S_s, where each subsystem S_s consists of a local scheduler along with a set of tasks. A system have a set of globally shared resource (lockable by any task in the system), and each subsystem has a set of local shared resource (only lockable by tasks in that subsystem).

7.4.1 Subsystem Model

For each subsystem S_s is specified by a subsystem (a.k.a. server) timing interface $S_s = \langle P_s, Q_s, p_s, B_s, X_s \rangle$, where P_s is the period and Q_s is the capacity allocated periodically to the subsystem where $0 < Q_s \leq P_s$ and X_s is the maximum execution-time that any subsystem-internal task may lock a shared global resource. Each server S_s has a unique priority p_s and at each instant during run-time a remaining budget B_s.

It should be noted that X_s is used for schedulability analysis only and our HSRP-implementation does not depend on the availability of this attribute. In the rest of this paper, we use the term subsystem and server interchangeably.

7.4.2 Task Model

For hard real-time systems, we are considering a simple periodic task model represented by a set Γ of n number of tasks. Each task τ_i is represented as $\tau_i = \langle T_i, C_i, \rho_i, b_i \rangle$, where T_i denotes the period of task τ_i with worst-case execution time C_i, ρ_i as its priority, and b_i its worst case local blocking. b_i is the longest execution-time inside a critical section with a resource-ceiling equal to or higher than ρ amongst all lower priority task inside the server of τ_i. A task, τ_i has a higher priority than another task, τ_j, if $\rho_i > \rho_j$. For simplicity, the deadline for each task is equal to T_i.

7.4.3 Scheduling Policy

We are using a fixed-priority scheduling FPS at the both global and local level. FPS is the de-facto standard used in industry. For hard-real time analysis we assume unique priorities for each server and unique priorities for each task within a server. However, our implementation support shared priorities, which are then handled in FIFO order (both at global and local scheduling).

7.4.4 Design Considerations

Here we present the challenges and goals that our implementation should satisfy:

1. **The use of HSF with resource sharing and the overrun mechanism:** User should be able to make a choice for using the HSF with resource sharing or the simple HSF without using shared resources. Further, user should be able to make a choice for selecting one of the overrun mechanisms, BO, PO, or EO.

2. **Consistency with the FreeRTOS kernel and keeping its API intact:** To embed the legacy code easily within a server in a two-level HSF, and to get minimal changes of the legacy system, it will be good to match the design of implementation with the underlying FreeRTOS operating

system. To increase the usability and understandability of HSF implementation for FreeRTOS users, major changes should not be made in the underlying kernel.

3. **Managing local/global system ceilings:** To ensure the correct access of shared resources at both local and global levels, the local and global system ceilings should be updated properly upon the locking and unlocking of those resources.

4. **Enforcement:** Enforcing server execution even at it's budget depletion while accessing a global shared resource; its currently executing task should not be preempted and the server should not be switched out by any other higher priority server (whose `priority` is not greater than the `systemceiling`) until the task releases the resource.

5. **Calculating and deducting overrun time of a server for PO and EO:** In case of payback (PO and EO), the overrun time of the server should be calculated and deducted from the budget at the next server activation.

6. **Protection of shared data structures:** The shared data structures that are used to lock and unlock both local and global shared resources should be accessed in a mutual exclusive manner with respect to the scheduler.

7.5 Implementation

7.5.1 Support for Time-Triggered Periodic Tasks

Since we are following the periodic resource model [25], we need the periodic task behavior implemented within the operating system. Like many other real-time operating systems, FreeRTOS does not directly support the periodic task activation. We incorporated the periodic task activation as given in Figure 7.2. To do minimal changes in the underlying operating system and save memory, we add only one additional variable `readyTime` to the task TCB, that describes the time when the task will become ready. A user API `vTaskWaitforNextPeriod(period)` is implemented to activate the task periodically. The FreeRTOS delayed-task list used to maintain the periodic tasks when they are not ready and waiting for their next activation period to start. Since FreeRTOS uses ticks, period of the task is given in number of ticks.

```
// task function
while (TRUE) do {
    taskbody();
    vTaskWaitforNextPeriod(period);
end while
```

Figure 7.2: Pseudo-code for periodic task model implementation

7.5.2 Support for Legacy System

To implement legacy applications support in HSF implementation for the FreeRTOS users, we need to map the original FreeRTOS API to the new API, so that the user can run its old code in a subsystem within the HSF. A macro `configHIERARCHICAL_LEGACY` must be set in the config file to utilize legacy support. The user should rename the old `main()` function, and remove the `vTaskStartScheduler()` API from legacy code.

The legacy code is created in a separate server, and in addition to the server parameters like period, budget, priority, user also provides a function pointer of the legacy code (the old main function that has been renamed). `xLegacyServerCreate(period, budget, priority, *serverHandle, *functionPointer)` API is provided for this purpose. The function first creates a server and then creates a task called `vLegacyTask(*functionPointer)` that runs only once and performs the initialization of the legacy code (executes the old main function which create the initial set of tasks for the legacy application), and destroys itself. When the legacy server is replenished first time, all the tasks of the legacy code are created dynamically within the currently running legacy server and start executing.

We have adopted the original FreeRTOS `xTaskGenericCreate` function to provide legacy support. If `configHIERARCHICAL_SCHEDULING` and `configHIERARCHICAL_LEGACY` macros are set then `xServerTaskGenericCreate` function is called that creates the task in the currently executing server instead of executing the original code of `xTaskGenericCreate` function.

This implementation is very simple and easy to use, user only needs to rename old `main()`, remove `vTaskStartScheduler()` from legacy code, and use a single API to create the legacy server. It should be noted that the HSF guarantees separation between servers; thus a legacy non/soft real-time server (which e.g. is not analyzed for schedulability or not use predictable resource locking) can co-exists with hard real-time servers.

7.5.3 Support for Resource sharing in HSF

Here we describe the implementation details of the resource sharing in two-level hierarchical scheduling framework. We implement the local and global resource sharing as defined by Davis and Burns [3]. For local resource sharing SRP is used and for global resource sharing HSRP is used. Further all the three forms of overrun as given by Behnam *et al.* [16] are implemented. The resource sharing is activated by setting the macro `configGLOBAL_SRP` in the configuration file.

Support for SRP

For local resource sharing we implement SRP to avoid problems like priority inversions and deadlocks.
The data structures for the local SRP: Each local resource is represented by the structure `localResource` that stores the resource ceiling and the task that currently holds the resource as shown in Figure 7.3. The locked resources are stacked onto the `localSRPList`; the FreeRTOS list structure is used to implement the SRP stack. The list is ordered according to the resource ceiling, and the first element of list has the highest resource ceiling, and represents the local `system ceiling`.
The extended functionality of the local scheduler with SRP: The only functionality extended is the searching for the next ready task to execute. Now the scheduler selects a task to execute if the task has the highest priority among all the ready tasks and its priority is greater than the current system ceiling, otherwise the task that has locked the highest (top) resource in the `localSRPList` is selected to execute. The API list for the local SRP is provided in the Appendix.

Support for HSRP

HSRP is implemented to provide global resource sharing among servers. The resource sharing among servers at the global level can be considered the same as sharing local resources among tasks at the local level. The details are as follows:
The data structures for the global HSRP: Each global resource is represented by the structure `globalResource` that stores the global-resource

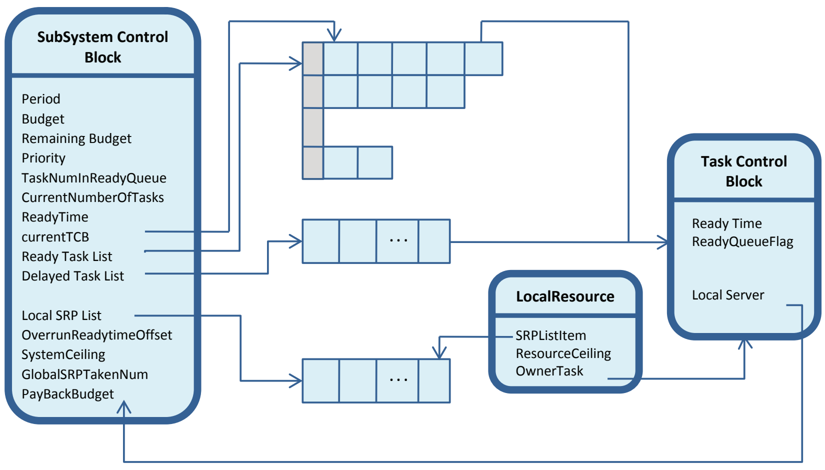


Figure 7.3: Data structures to implement SRP

ceiling and the server that currently holds the resource as shown in Figure 7.4. The locked resources are stacked onto the `globalHSRPList`; the FreeRTOS list structure is used to implement the HSRP stack. The list is ordered according to the resource ceiling, the first element of the list has the highest resource ceiling and represents the `GlobalSystemCeiling`.

The extended functionality of the global scheduler with HSRP: To incorporate HSRP into the global scheduler, `prvChooseNextIdlingServer()` and `prvChooseNextDeferrableServer()` macros are appended with the following functionality: The global scheduler selects a server if the server has the highest priority among all the ready servers and the server's priority is greater than the current `GlobalSystemCeiling`, otherwise the server that has locked the highest(top) resource in the `HSRPList` is selected to execute. The API list for the global HSRP is provided in Appendix.

Support for Overrun Protocol

We have implemented three types of overrun mechanisms; without payback (BO), with payback (PO), and enhanced overrun (EO). Implementation of BO is very simple, the server simply executes and overruns its budget, and no further action is required. For PO and EO we need to measure the overrun amount

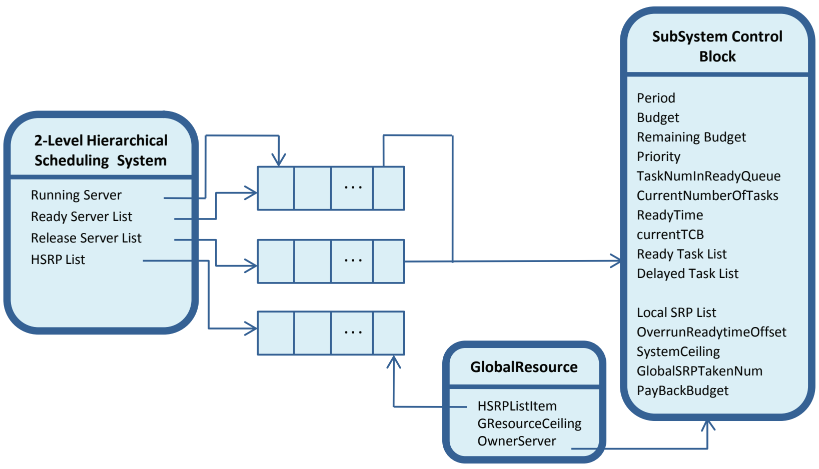


Figure 7.4: Data structures to implement HSRP

of time to pay back at the server's next activation.

The data for the PO and EO overrun mechanisms: Two variables `PayBack Budget` and `OverrunReadytimeOffset` are added to the subsystem structure `subSCB` to keep a record of the overrun amount to be deducted from the next budget of the server as shown in Figure 7.4. The overrun time is measured and stored in `PayBackBudget`. `OverrunReadytimeOffset` is used in EO mechanism to impose an offset in the next activation of server.

The extended functionality of the global scheduler with overrun: A new API `prvOverrunAdjustServerNextReadyTime(*pxServer)` is used to embed overrun functionality (PO and EO) into the global scheduler. For both PO and EO, the amount of overrun, i.e. `PayBackBudget` is deducted from the server `RemainingBudget` at the next activation period of the server, i.e. $B_s = Q_s - \theta_s$. For EO, in addition to this, an offset (O_s) is calculated that is equal to the amount of overrun, i.e. $O_s = \theta_s$. The server's next activation time (the budget replenishment of subsystem) is delayed by this offset. `OverrunReadytimeOffset` variable is used to store the offset for next activation of the server.

Safety Measure

We have modified `vTaskDelete` function in order to prevent the system from crashing when users delete a task which still holds a local SRP or a global HSRP resource. Now it also executes two private functions `prvRemoveLocalResourceFromList (*pxTaskToDelete)`, and `prvRemoveGlobalResourceFromList (*pxTaskToDelete)`, before the task is deleted.

7.5.4 Addressing Design Considerations

Here we address how we achieve the design requirements that are presented in Section 7.4.4.

1. **The use of HSF with resource sharing and the overrun mechanism:** The resource sharing is activated by setting the macro `configGLOBAL_SRP` in the configuration file. The type of overrun can be selected by setting the macro `configOVERRUN_PROTOCOL_MODE` to one of the three values: `OVERRUN_WITHOUT_PAYBACK`, `OVERRUN_PAYBACK`, or `OVERRUN_PAYBACK_ENHANCED`.

2. **Consistency with the FreeRTOS kernel and keeping its API intact:** We have kept consistence with the FreeRTOS from the naming conventions to API, data structures and the coding style used in our implementations; for example all the lists used in our implementation are maintained in a similar way as of FreeRTOS.

3. **Managing local/global system ceilings:** The correct access of the shared resources at both local and global levels is implemented within the functionality of the API used to lock and unlock those resources.

   When a task locks a local/global resource whose ceiling is higher than the subsystem/system ceiling, the resource mutex is inserted as the first element onto the `localSRPList`/`HSRPList`, the `systemceiling` /`GlobalSystemCeiling` is updated, and this task/server becomes the owner of this local/global resource respectively. Each time a global resource is locked, the `GlobalResourceTakenNum` is incremented.

   Similarly upon unlocking a local/global resource, that resource is simply removed from the top of the `localSRPList`/`HSRPList`, the `systemceiling` /`GlobalSystemCeiling` is updated, and the owner of this resource is set to `NULL`. For global resource, the `GlobalResourceTakenNum` is decremented.

4. **Enforcement:** `GlobalResourceTakenNum` is used as an overrun flag, and when its value is greater than zero (means a task of the currently executing server has locked a global resource), no other higher priority server (whose `priority` is not greater than the `systemceiling`) can preempt this server even if its budget depletes.

5. **Overrun time of a server for PO and EO:** `prvOverrunAdjustServer NextReadyTime` API is used to embed the overrun functionality into the global scheduler as explained in section 7.5.3.

6. **Protection of shared data structures:** All the functionality of the APIs (for locking and unlocking both local and global shared resources) is executed within the FreeRTOS macros `portENTER_CRITICAL()` and `port EXIT_CRITICAL()` to protect the shared data structures.

7.6 Schedulability analysis

This section presents the schedulability analysis of the HSF, starting with local schedulability analysis (i.e. checking the schedulability of each task within a server, given the servers timing interface), followed by global schedulability analysis (i.e., checking that each server will receive its capacity within its period given the set of all servers in a system).

7.6.1 The Local Schedulability Analysis

The local schedulability analysis can be evaluated as follows [25]:

$$\forall \tau_i \ \exists t : 0 < t \leq D_i, \ \ \texttt{rbf}(i,t) \leq \texttt{sbf}(t), \tag{7.1}$$

where `sbf` is the supply bound function, based on the periodic resource model presented in [25], that computes the minimum possible CPU supply to S_s for every time interval length t, and $\texttt{rbf}(i,t)$ denotes the request bound function of a task τ_i which computes the maximum cumulative execution requests that could be generated from the time that τ_i is released up to time t and is computed as follows:

$$\texttt{rbf}(i,t) = C_i + b_i + \sum_{\tau_k \in \texttt{HP}(i)} \left\lceil \frac{t}{T_k} \right\rceil \cdot C_k, \tag{7.2}$$

where $\texttt{HP}(i)$ is the set of tasks with priorities higher than that of τ_i and b_i is the maximum local blocking.

The evaluation of $\texttt{sbf}$ depends on the type of the overrun mechanism;

Overrun without payback

$$\texttt{sbf}(t) = \begin{cases} t - (k+1)(P_s - Q_s) & \text{if } t \in W^{(k)} \\ (k-1)Q_s & \text{otherwise,} \end{cases} \tag{7.3}$$

where $k = \max\Big(\big\lceil\big(t - (P_s - Q_s)\big)/P_s\big\rceil, 1\Big)$ and $W^{(k)}$ denotes an interval $[(k+1)P_s - 2Q_s, (k+1)P_s - Q_s]$.

Overrun with payback [16]

$$\texttt{sbf}(t) = \max\Big(\min\Big(f_1(t), f_2(t)\Big), 0\Big), \tag{7.4}$$

where $f_1(t)$ is

$$f_1(t) = \begin{cases} t - (k+1)(P_s - Q_s) - X_s & \text{if } t \in W^{(k)} \\ (k-1)Q_s & \text{otherwise,} \end{cases} \tag{7.5}$$

where $k = \max\Big(\big\lceil\big(t - (P_s - Q_s) - X_s\big)/P_s\big\rceil, 1\Big)$ and $W^{(k)}$ denotes an interval $[(k+1)P_s - 2Q_s + X_s, (k+1)P_s - Q_s + X_s]$, and $f_2(t)$ is

$$f_2(t) = \begin{cases} t - (2)(P_s - Q_s) & \text{if } t \in V^{(k)} \\ t - (k+1)(P_s - Q_s) - X_s & \text{if } t \in Z^{(k)} \\ (k-1)Q_s - X_s & \text{otherwise,} \end{cases} \tag{7.6}$$

where $k = \max\Big(\big\lceil\big(t - (P_s - Q_s)\big)/P_s\big\rceil, 1\Big)$, $V^{(k)}$ denotes an interval $[2P_s - 2Q_s, 2P_s - Q_s - X_s]$, and $Z^{(k)}$ denotes an interval $[(k+2)P_s - 2Q_s, (k+2)P_s - Q_s]$.

Enhanced overrun

$$\texttt{sbf}(t) = \max\Big(f_2(t), 0\Big). \tag{7.7}$$

7.6.2 The Global Schedulability Analysis

A global schedulability condition is

$$\forall S_s \ \exists t : 0 < t \leq P_s, \ \ \mathtt{RBF}_s(t) \leq \mathtt{t}. \tag{7.8}$$

where $\mathtt{RBF}_s(t)$ is the request bound function and it is evaluated depending on the type of server (deferrable or idling) and type of the overrun mechanism (see [16] for more details). First, we will assume the idling server and later we will generalize our analysis to include deferrable server.

Overrun without payback

$$\mathtt{RBF}_s(t) = (Q_s + X_s + Bl_s) + \sum_{S_k \in \mathtt{HPS}(s)} \left\lceil \frac{t}{P_k} \right\rceil \cdot (Q_k + X_k). \tag{7.9}$$

where $\mathtt{HPS}(s)$ is the set of subsystems with priority higher than that of S_s. Let Bl_s denote the maximum blocking imposed to a subsystem S_s, when it is blocked by lower-priority subsystems.

$$Bl_s = \max\{X_j | \ S_j \in \mathtt{LPS}(S_s)\}, \tag{7.10}$$

where $\mathtt{LPS}(S_s)$ is the set of subsystems with priority lower than that of S_s.

Overrun with payback

$$\mathtt{RBF}_s(t) = (Q_s + X_s + Bl_s) + \sum_{S_k \in \mathtt{HPS}(s)} \left(\left\lceil \frac{t}{P_k} \right\rceil (Q_k) + X_k \right). \tag{7.11}$$

Enhanced overrun

$$\mathtt{RBF}_s(t) = (Q_s + X_s + Bl_s) + \sum_{S_k \in \mathtt{HPS}(s)} \left(\left\lceil \frac{t + J_k}{P_k} \right\rceil (Q_k) + X_k \right). \tag{7.12}$$

Where $J_s = X_s$ and the schedulability analysis for this type is

$$\forall S_s, 0 < \exists t \leq P_s - X_s, \ \ \mathtt{RBF}_s(t) \leq \mathtt{t}, \tag{7.13}$$

For deferrable server, a higher priority server may execute at the end of its period and then at the beginning of the next period. To model such behavior a jitter (equal to $P_k - (O_k + X_k)$) is added to the ceiling in equations 7.9, 7.11 and 7.12.

7.6.3 Implementation Overhead

In this section we will explain how to include the implementation overheads in the global schedulability analysis.

Looking at the implementation we can distinguish two types of runtime overhead associated with the system tick: (1) a repeated overhead every system tick independently if it will release a new server or not, and (2) an overhead which occurs whenever a server is activated and it includes the overhead of scheduling, maybe context switch, budget depletion after consuming the budget then another context switch and scheduling and finally it includes the overrun overhead.

(1) Is called fixed overhead (fo) and it is the result of updating the system tick and perform some checking and its value is always fixed. This overhead can be added to equation 7.8. This equation assumes that the processor can provide all CPU time to the servers (t in the right side of the equation) now we assume that every system tick (st), a part will be consumed by the operating system (fo) and then instead of using t in the right side of equation 7.8, we can use $(1 - fo/st) \times t$ to include the fixed overhead. (st defaults to $1ms$ for our implementation.)

(2) Is called server overhead (so) and repeats periodically for every server, i.e. with a period P_i. Since the server overhead is executed by the kernel its not enough to model it as extra execution demand from the server. Instead the overhead should be modeled as a separate server S_o (one server S_o corresponding to each real server S_i) executing at a priority higher that of any real server with parameters $P_o = P_i$, $Q_o = so$, and $X_o = 0$.

The overhead-parameters are dependent on the number of servers, tasks and priority levels, etc. and should be quantified with static WCET-analysis which is beyond the scope of this paper; however some small test cases reported in [8] the measured worst-case for idling servers are $fo = 32\mu s$ and $so = 74\mu s$, and for deferrable servers they are $fo = 32\mu s$ and $so = 85\mu s$ for three servers with total seven tasks.

7.7 Experimental Evaluation

In this section, we report the evaluation of behavior and performance of the resource sharing in HSF implementation. All measurements are performed on the target platform EVK1100 [9]. The AVR32UC3A0512 micro-controller runs at the frequency of 12MHz and its tick interrupt handler at 1ms.

7.7.1 Behavior Testing

In this section we perform an experiment to test the behavior of overrun in case of global resource sharing in HSF implementation. The experiment is performed to check the overrun behavior in idling periodic server by means of a trace of the execution. Two servers S1, and S2 are used in the system, plus idle server is created. The servers used to test the system are given in Table 8.1.

Server	S1	S2
Priority	2	1
Period	20	40
Budget	10	15

Table 7.1: Servers used to test system behavior.

Note that higher number means higher priority. Task properties and their assignments to the servers is given in Table 8.2. $T2$ and $T3$ share a global resource. The execution time of $T2$ is $(3 + 3)$ that means a normal execution for initial 3 time units and the critical section execution for the next 3 time units, similarly $T3$ $(10 + 8)$ executes for 10 time units before critical section and executes for 8 time units within critical section. The visualization of the executions of budget overrun without payback (BO) and with payback (PO) for idling periodic server are presented in Figure 7.5 and Figure 7.6 respectively.

Tasks	T1	T2	T3
Servers	$S1$	$S1$	$S2$
Priority	2	1	1
Period	15	20	60
Execution Time	3	$(3+3)$	$(10+9)$

Table 7.2: Tasks in both servers.

In the visualization, the arrow represents task arrival, a gray rectangle means task execution. In Figure 7.5 at time 20, the high priority server $S1$ is replenished, but its priority is not higher than the global system ceiling, therefore, it cannot preempt server $S2$ which is in the critical section. $S2$ depletes its budget at time 25, but continues to executes in its critical section until it unlocks the global resource at time 29. The execution of S1 is delayed by 9 time units.

In case of overrun with payback, the overrun time is deducted from the

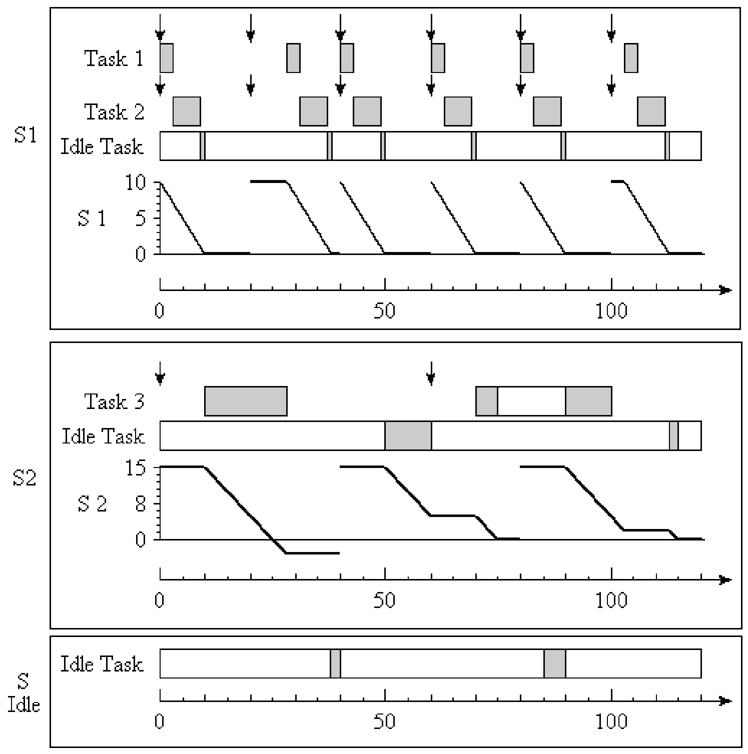


Figure 7.5: Trace of budget overrun without payback (BO) for Idling server

budget at the next server activation, as shown in Figure 7.6. At time 40 the server $S2$ is replenished with a reduced budget, while in case of overrun without payback the server is always replenished with its full budget as obvious from Figure 7.5.

7.7.2 Performance Measures

Here we report the performance measures of lock and unlock functions for both global and local shared resources.

The execution time of functions to lock and unlock global and local resources is presented in Table 7.3. For each measure, a total of 1000 values are computed. The minimum, maximum, average and standard deviation on these

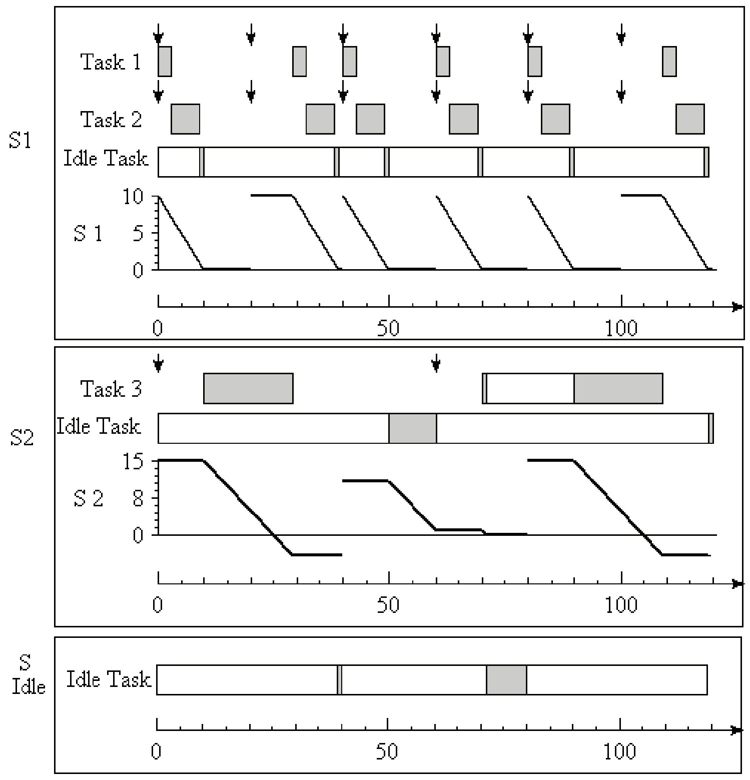


Figure 7.6: Trace of budget overrun with payback (PO) for Idling server

values are calculated and presented for both types of resource sharing.

7.8 Conclusions

In this paper, we have provided a hard real-time support for a two-level HSF implementation in an open source real-time operating system FreeRTOS. We have implemented the periodic task model within the FreeRTOS kernel. We have provided a very simple and easy implementation to execute a legacy system in the HSF with the use of a single API. We have added the SRP to the FreeRTOS for efficient resource sharing by avoiding deadlocks. Further we

Function	Min.	Max.	Average	St. Dev.
`vGlobalResourceLock`	21	21	21	0
`vGlobalResourceUnlock`	32	32	32	0
`vLocalResourceLock`	21	32	26.48	5.51
`vLocalResourceUnlock`	21	21	21	0

Table 7.3: The execution time $(\text{in micro-seconds } (\mu\text{s}))$ of global and local lock and unlock function.

implemented HSRP and overrun mechanisms (BO, PO, EO) to share global resources in a two-level HFS implementation. Under assumption of nested locking, the overrun is bounded and is equal to the longest resource-holding time. Hence, the temporal isolation of HSF is subject to the bounded resource-holding time.

We have focused on doing minimal modifications in the kernel to keep the implementation simple and keeping the original FreeRTOS API intact. We have presented the design and implementation details and have tested our implementations on the EVK1100 board. We have included the overheads for local-level and global-level resource sharing into the schedulability analysis. In future we plan to integrate the virtual node concept of ProCom model ontop of the presented HSF [6, 7].

7.9 Appendix

A synopsis of the application program interface to implement resource sharing in HSF implementation is presented below. The names of these API are self-explanatory.

1. `xLocalResourcehandle xLocalResourceCreate(uxCeiling)`
2. `void vLocalResourceDestroy(xLocalResourcehandle)`
3. `void vLocalResourceLock(xLocalResourcehandle)`
4. `void vLocalResourceUnLock(xLocalResourcehandle)`
5. `xGlobalResourcehandle xGlobalResourceCreate (uxCeiling)`
6. `void vGlobalResourceDestroy(xGlobalResourcehandle)`
7. `void vGlobalResourceLock(xGlobalResourcehandle)`
8. `void vGlobalResourceUnLock(xGlobalResourcehandle)`

Bibliography

[1] Z. Deng and J. W.-S. Liu. Scheduling real-time applications in an open environment. In *Proc. 18th IEEE Real-Time Systems Symposium (RTSS)*, 1997.

[2] G. Lipari, P. Gai, M. Trimarchi, G. Guidi, and P. Ancilotti. A hierarchical framework for component-based real-time systems. *Component-Based Software engineering*, LNCS-3054(2004):209–216, May 2005.

[3] R. I. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *IEEE Real-Time Systems Symposium (RTSS'06)*, pages 389–398, 2006.

[4] M. Behnam, I. Shin, T. Nolte, and M. Nolin. SIRAP: a synchronization protocol for hierarchical resource sharing in real-time open systems. In *ACM & IEEE conference on Embedded software (EMSOFT'07)*, pages 279–288, 2007.

[5] FreeRTOS web-site. http://www.freertos.org/.

[6] Jan Carlson, Juraj Feljan, Jukka Mäki-Turja, and Mikael Sjödin. Deployment Modelling and Synthesis in a Component Model for Distributed Embedded Systems. In *Proceedings of the 36th Euromicro Conference on Software Engineering and Advanced Applications (SEAA' 10)*, September 2010.

[7] Rafia Inam, Jukka Mäki-Turja, Jan Carlson, and Mikael Sjödin. Using temporal isolation to achieve predictable integration of real-time components. In *22nd Euromicro Conference on Real-Time Systems (ECRTS10) WiP Session*, pages 17–20, July 2010.

[8] Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, S. M. H. Ashjaei, and Sara Afshar. Hierarchical scheduling framework implementation in freertos. Technical Report, Mälardalen University, April 2011.

[9] ATMEL EVK1100 product page. http://www.atmel.com/dyn/Products/tools_card.asp?tool_id=4114.

[10] Daeyoung Kim, Yann-Hang Lee, and M. Younis. Spirit-ukernel for strongly partitione real-time systems. In *Proc. of the 7th International conference on Real-Time Computing Systems and Applications (RTCSA'00)*, 2000.

[11] T. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems*, 3(1):67–99, 1991.

[12] S. Saewong and R. Rajkumar. Hierarchical reservation support in resource kernels. In *Proc. 22th IEEE Real-Time Systems Symposium (RTSS)*, 2001.

[13] G. Buttazzo and P. Gai. Efficient edf implementation for small embedded systems. In *International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'06)*, 2006.

[14] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *Journal of IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[15] N. Fisher, M. Bertogna, and S. Baruah. The design of an edf-scheduled resource-sharing open environment. In *IEEE Real-Time Systems Symposium(RTSS'07)*, pages 83–92, 2004.

[16] Moris Behnam, Thomas Nolte, Mikael Sjödin, and Insik Shin. Overrun Methods and Resource Holding Times for Hierarchical Scheduling of Semi-Independent Real-Time Systems. *IEEE Transactions on Industrial Informatics*, 6(1), February 2010.

[17] Moris Behnam, Thomas Nolte, Insik Shin, Mikael Åsberg, and Reinder J. Bril. Towards hierarchical scheduling on top of vxworks. In *Proceedings of the Fourth International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'08)*, pages 63–72, July 2008.

[18] Mikael Åsberg, Moris Behnam, Thomas Nolte, and Reinder J. Bril. Implementation of Overrun and Skipping in VxWorks. In *Proceedings of the 6th International Workshop (OSPERT10)*, 2010.

[19] Mike Holenderski, Wim Cools, Reinder J. Bril, and J. J. Lukkien. Extending an Open-source Real-time Operating System with Hierarchical Scheduling. Technical Report, Eindhoven University, 2010.

[20] Mike Holenderski, Wim Cools, Reinder J. Bril, and J. J. Lukkien. Multiplexing Real-time Timed Events. In *Work in Progress session of the IEEE International Conference on Emerging Techonologies and Factory Automation (ETFA09)*, 2009.

[21] M.M.H.P. van den Heuvel, Mike Holenderski, Wim Cools, Reinder J. Bril, and Johan J. Lukkien. Virtual Timers in Hierarchical Real-time Systems. In *Work in Progress Session of the IEEE Real-Time Systems Symposium (RTSS09)*, December 2009.

[22] M.M.H.P. van den Heuvel, M. Holenderski, W. Cools, R. J. Bril, and J. J. Lukkien. Extending an HSF-enabled Open-Source Real-Time Operating System with Resource sharing. In *(OSPERT10)*, 2010.

[23] Microchip web-site.

[24] EE TIMES web-site. http://www.eetimes.com/design/embedded/4008920/The-results-for-2010-are-in-.

[25] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proc. 24th IEEE Real-Time Systems Symposium (RTSS)*, pages 2–13, 2003.

Chapter 8

Paper D: Run-Time Component Integration and Reuse in Cyber-Physical Systems

Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, Jiří Kunčar
MRTC report ISSN 1404-3041 ISRN MDH-MRTC-256/2011-1-SE,
Mälardalen University, December, 2011.
In submission for conference publishing (ICCPS' 12).

Abstract

We present the concept of runnable virtual nodes as a means to achieve predictable integration and reuse of software components in cyber-physical systems. A runnable virtual node is a coarse-grained real-time component that provides functional and temporal isolation with respect to its environment. Its interaction with the environment is bounded both by a functional and a temporal interface, and the validity of its internal temporal behavior is preserved when integrated with other components or when reused in a new environment. Our realization of runnable virtual nodes exploits the latest techniques for hierarchical scheduling to achieve temporal isolation, and the principles from component-based software-engineering to achieve functional isolation. In the report we present a proof-of-concept case study, implemented in the ProCom component-technology executing on top of FreeRTOS based hierarchical scheduling framework.

8.1 Introduction

A contemporary Cyber-Physical System (CPS) is often required to monitor and control several disparate variables in its environment. From a development point of view, it often makes sense to develop the different control-functions as separate software-components [1]. Typically, these components are first developed and tested in isolation, and later integrated to form the final software for the system. Furthermore, many industrial systems are developed in an evolutionary fashion, reusing components from previous versions or from related products. It means that the reused components are re-integrated in new environments.

When multiple components are deployed on the same hardware node, the emerging timing behavior of each of the components is typically unpredictable. For a cyber-physical system with real-time constraints, this means that a component that is found correct during unit testing may fail, due to a change in temporal behavior, when integrated in a system. Even if a new component is still operating correctly in the system, the integration could cause a previously integrated (and correctly operating) component to fail. Similarly, the temporal behavior of a component is altered if the component is reused in a new system. Since this alteration is unpredictable as well, a previously correct component may fail when reused.

While using the temporal models of component behavior and requirements, some of the problems may be mitigated by using scheduling analysis [2, 3], however these techniques only allow very simple models; typically simple timing attributes such as period and deadline are used. In industry, components often exhibit a too complex behavior to be amenable for the scheduling analysis. And, even if a suitable analysis technique should exist, such analysis requires knowledge of the temporal behavior of all components in the system. Thus, a component cannot be deemed correct without knowing which components it will be integrated with. Further the reuse of a component is restricted since it is very difficult to know beforehand if the component will pass a schedulability test in a new system.

For complex real-time CPS, methodologies and techniques are required to provide not only functional isolation but also temporal isolation so that the runtime timing properties could be guaranteed. Further the real-time properties of the components should be maintained for their reuse in large-scale industrial CPS.

To remedy this situation we propose the concept of a *runnable virtual node*, which is an execution-platform concept that preserves temporal properties of the software executed in the virtual node [4]. It introduces an intermediate level between the functional entities and the physical nodes. Thereby it leads to a *two-level deployment process* instead of a single big-stepped deployment; i.e. deploying functional entities to the virtual nodes and then deploying virtual nodes to the physical nodes.

The virtual node is intended for coarse-grained components for single node deployment and with potential internal multitasking. The idea is to encapsulate the real-time properties into model-driven reusable components-based systems to achieve not only the predictable integrations and reusability of those components [4, 5] but also maintenance, testing, and extendibility. To achieve this, the timing properties of the components should be preserved so that real-time components integration and reuse can be made predictable.

Hierarchical Scheduling Framework (HSF) [6, 7] is known as a technique for providing temporal isolation between applications in the real-time community. Recently, HSF is proposed to develop complex CPS by enabling temporal isolation and predictable resource usage of CPS software [8]. In this report, we integrate HSF within a component technology for embedded real-time systems; to realize our ideas of guaranteeing temporal properties of real-time components, their predictable integrations and reusability. We introduce the runnable virtual node, which includes the executable representation of the components (i.e. a set of tasks), a resource allocation, and a real-time scheduler to be executed within a server in the HSF. The server executes with a guaranteed temporal behavior, using its allocated CPU bandwidth, regardless of any other execution on the physical node. Thus, once a server has been configured for the virtual node, its real-time properties will be preserved when the virtual node is integrated with other virtual nodes on a physical node, or when a virtual node is reused in another context.

Contributions

The work presented in this report is within the context of ProCom component technology [9]. The main contributions are as follows:

- We *realize the concept of runnable virtual nodes* by embedding the HSF implementation within the ProCom component technology for an embedded platform running FreeRTOS operating system [10].
- We *introduce a two-level deployment process* instead of a single big-stepped deployment; i.e. deploying functional entities to the virtual nodes and then deploying virtual nodes to the physical nodes, thereby

preserving the timing properties within the components in addition to their functional properties.

- We *provide a case study* as a proof of concept and run it on an AVR 32-bit board EVK1100 [11].
- We *test the runnable virtual node's real-time properties* for temporal isolations and reusability.

Outline: Section 8.2 presents the related work on component-based and model-driven component systems. Section 8.3 gives an overview about the ProCom Component Model and the HSF implementation. In Section 8.4, we describe the runnable virtual node and how it is used within the ProCom technology. Section 8.5 presents a case-study in which runnable virtual nodes have been used. Finally, Section 8.6 concludes the article.

8.2 Related Work

In this section, we describe some contemporary component-technologies available for embedded systems. Especially we focus on the deployment and integration of components and on the predictability in the time-domain of the resulting systems. The list is by far non-exhaustive, but rather focuses on state-of-the-art within industrial applications. In the academic domain several similar concepts exist; but to the best of our knowledge, no other technology employs a two-phase deployment process or employs HSF to support temporal isolation, ease of integration or component reuse.

8.2.1 AUTOSAR

AUTomotive Open System ARchitecture (AUTOSAR) [12] is an open standard for automotive electronics architectures. It is developed to deal with the increasing complexity and to fulfil a number of future vehicle requirements (such as safety, availability, driver assistance, infotainment etc.). The key features of AUTOSAR are modularity, configurability, standardized interfaces and a runtime environment.

Functional software is developed using component-based approach [13]. A component is developed over many layers of AUTOSAR, including: Application layer, Runtime Environment (RTE), Basic software and ECU (Electronic Control Unit, is a node in an automotive network) hardware. A Software

Component (SW-C) at ECU level contains at least one or several runnable entities. A runnable is a small fragment of sequential code within a component [14]. In AUTOSAR, the deployment begins when the RTE generator maps all runnables to the OS tasks and builds inter- ECU and intra-ECU communications among them. After mapping, the RTE generator configures each ECU and constructs the OS tasks bodies.

A main disadvantage of AUTOSAR is that it lacks clear and well-defined timing properties that further affect the execution semantics too. On the other hand, ProCom puts special focus on such requirements right from the beginning of the component's development untill the component's deployment. The runnable virtual nodes of ProCom are reusable components preserving timing properties in them and are independent of the platform specifics. As compared to AUTOSAR, ProCom clearly distinguishes between the data and control flows among ProCom components. Deployment in AUTOSAR is a single-stepped process and the executables are generated directly from the components, unlike ProCom where the deployment is performed in two steps at both modelling and synthesis levels. A tool suite to support the complete AUTOSAR methodology for hard real-time systems is still missing.

8.2.2 Rubus

The Rubus Component Model (RCM) [15] is developed in cooperation between Arcticus Systems [16] and Mälardalen University. It is used commercially in the automotive industry. In many aspects RCM is similar to the ProCom: for example rubus captures the functionality at two-levels of component hierarchy; at first level the components are passive while at the second level they are active; manages different ports for control and data flows. The component technology uses a graphical design tool and a scheduler for system design, and some plug-ins to perform analysis. Finally, the run-time infrastructure is generated using Execution Models (EM) for the desired platform.

In Rubus, the real-time requirements of the components are realized by the use of EMs, which are logical objects and are defined in the infrastructure at the run-time environment. The RCM is not restricted to the use of a specific run-time environment as long as the components preserve the semantics defined in them. However, the current task set can only be executed in the RubusRTOS [17].

Its main difference from the ProCom technology is at the deployment and at the execution levels. In Rubus, the deployment is a single-stepped process for the desired platform. On the other hand, in Procom, the deployment is a two-stepped process. In Rubus, the required hardware components are directly modeled in the Rubus components, unlike ProCom where the components are developed independently from the hardware details and the hardware specifics are taken care at the last step of the deployment process. Another difference is the platform dependence of the Rubus technology on the underlying operating system [15]. The platform on which the component has to be executed is modeled within the Rubus components and the final executables are generated only for that particular platform.

8.2.3 AADL

Architecture Analysis and Design Language (AADL) was developed as a SAE Standard AS-5506 [18] to design and analyze software and hardware architectures of distributed real-time embedded systems. Modeling of software and hardware parts is supported by software components and execution platform components respectively [19]. Properties and new functional aspects can be attached to the elements (e.g., components, connections) using the properties defined in the SAE standard, and communication among components is performed using component interfaces i.e., ports. Ocarina [20] is a tool suite that facilitates the design of AADL models and their mapping on a hardware platform, assessment of these models, automatic code generation, and deployment.

Deployment in AADL is supported by a middleware API called PolyORB (PolyORB for code generation in Ada while PolyORB-HI for code generation in C). Runnable entities are presented by processes. A process contains many tasks and it is a self-contained runnable entity that executes on a hardware platform.

Unlike ProCom, the deployment in AADL is done in a single step to directly execute the generated code on the physical platform. Another type of runnable entity for AADL employs a hierarchical scheduling concept in a partition. A partition is a combination of several processes and a scheduler called Virtual Processor [21]. But this kind of runnbale entity is also deployed in a single step. It provides temporal partitioning like runnable virtual node of ProCom, but it does not consider the reusability aspect of the runnable components.

8.2.4 Deployment and Configuration specification

The Deployment and Configuration specification (D&C) [22] is standardized by the Object management Group. Its main purpose is to facilitate the deployment of component-based applications onto target platform. It uses a Platform Independent Model (PIM) for the model components with three level, and a Platform Specific Model (PSM) for the CORBA Component Model (CCM).

Recently, an extension has been proposed to support the development of applications with real-time properties and provide a deployment plan, called RT-D&C [23]. The metadata about the temporal behaviour of components is added to the specification at the PIM level to facilitate the real-time analysis of the components. However, a RT-planner, who configures the real-time application after using the real-time analysis tools, is required to assign the timing properties to the application which is different from the ProCom deployment. As compared to ProCom where the timing properties are preserved within the runnable virtual nodes, RT-D&C only preserves them till the components development at PIM level.

8.3 Background

This section presents the background technologies our work uses. We provide an overview of the ProCom component technology, followed by and introduction of the Hierarchical Scheduling Framework, and its implementation in FreeRTOS.

8.3.1 ProCom Component Model

Component-Based Software Engineering (CBSE) and Model-Based Engineering (MBE) are two emerging approaches to develop embedded control systems like software used in trains, airplanes, cars, industrial robots, etc. The ProCom component technology combines both CBSE and MBE techniques for the development of the system parts, hence also exploits the advantages of both. It takes advantages of encapsulation, reusability, and reduced testing from CBSE. From MBE, it makes use of automated code generation and performing analysis at an earlier stage of development. In addition, ProCom achieves additional benefits of combining both approaches (like flexible reuse, support for mixed maturity, reuse and efficiency tradeoff) [4].

The ProCom component model can be described in two distinct realms: the modeling and the runnable realms as shown in Figure 8.1. In Modeling realm, the models are made using CBSE and MBE while in runnable realm, the synthesis of runnable entities is done from the model entities. Both realms are explained as follows:

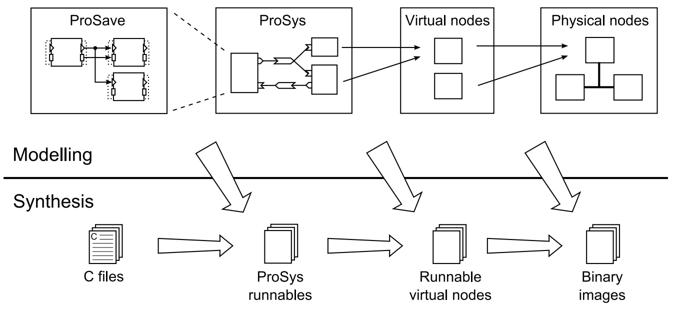


Figure 8.1: An overview of the deployment modelling formalisms and synthesis artefacts.

The Modeling Realm

Modeling in ProCom is done by four discrete but related formalisms as shown in Figure 1. The first two formalisms relate to the system functionality modeling while the later two represent the deployment modeling of the system. Functionality of the system is modeled by the ProSave and ProSys components at different levels of granularity. The basic functionality (data and control) of a simple component is captured in ProSave component level, which is passive in nature. At the second formalism level, many ProSave components are mapped to make a complete subsystem called ProSys that is active in nature. Both ProSys and ProSave allow composite components. For details on ProSave and ProSys, including the motivation for separating the two, see [24, 9].

The deployment modeling is used to capture the deployment related design decisions and then mapping the system to run on the physical platform. Many ProSys components can be mapped together on a virtual node (many-to-one mapping) together with a resource budget required by those components. After that many virtual nodes could be mapped on a physical node i.e. an ECU. The relationship is again many-to-one. This part represents all the physical nodes, their intercommunication through the network and the type of the network etc. Details about the deployment modeling are provided in [4].

The Runnable (or Executable) Realm

is the synthesis of runnables/executables from the ProCom model entities. The primitive ProSave components are represented as a simple C language source code in runnable form. From this C code, the ProSys runnables are generated which contain the collection of operating system tasks. Virtual node runnables implement the local scheduler and contain the tasks in a server. Hence a runnable virtual node actually encapsulates the set of tasks, resource allocations, and a real-time scheduler within a server in a two-level hierarchical scheduling framework. Final binary image is generated by connecting different virtual nodes together with a global scheduler and using the middleware to provide intra-communications among the virtual node executables.

Deployment and Synthesis Activities

Rather than deploying a whole system in one big step, the deployment of the ProCom components on the physical platform is done in the following two steps:

- First the ProSys subsystems are deployed on an intermediate node called virtual node. The allocation of ProSys subsystems to the virtual nodes is many-to-one relationship. The additional information that is added at this step is the resource budgets (CPU time).
- The virtual nodes are then deployed on the physical nodes. The relationship is again many-to-one, which means that more than one virtual node can be deployed to one physical node.

This two-steps deployment process allows not only the detailed analysis in isolation from the other components to be deployed on the same physical node, but once checked for correctness, it also preserves its temporal properties for further reuse of this virtual node as an independent component. Section 8.4 describes this further.

The PRIDE tool supports the automatic synthesis of the components at different levels [25]. At the ProSave level, the XML descriptions of the components is the input and the C files are generated containing the basic functionality. At the second level, ProSys components are assigned to the tasks to generate ProSys runnables. Since the tasks at this level are independent of the execution platform, therefore, the only attribute assigned at this stage is the period for each task; which they get from the clock frequency that is triggering the specific component. A clock defines the periodic triggering of components with a specified frequency. Components are allocated to a task when (i) the components are triggered by the same event, (ii) when the components have precedence relation among them to be preserved. The synthesis of the runnable virtual nodes and final executables is given in Section 8.4.2.

8.3.2 Hierarchical Scheduling Framework

A two-level Hierarchical Scheduling Framework (HSF) [6] is used to provide the temporal isolation among a set of subsystems. In hierarchical scheduling, the CPU time is partitioned among many subsystems (or servers), that are scheduled by a global (system-level) scheduler. Each server contains its own internal set of tasks that are scheduled by a local (subsystem-level) scheduler. Hence a two-level HSF can be viewed as a tree with one parent node (global scheduler) and many leaf nodes (local schedulers) as illustrated in Figure 8.2. The parent node is a global scheduler that schedules subsystems. Each subsystem has its own local scheduler, that schedules the tasks within the subsystem.

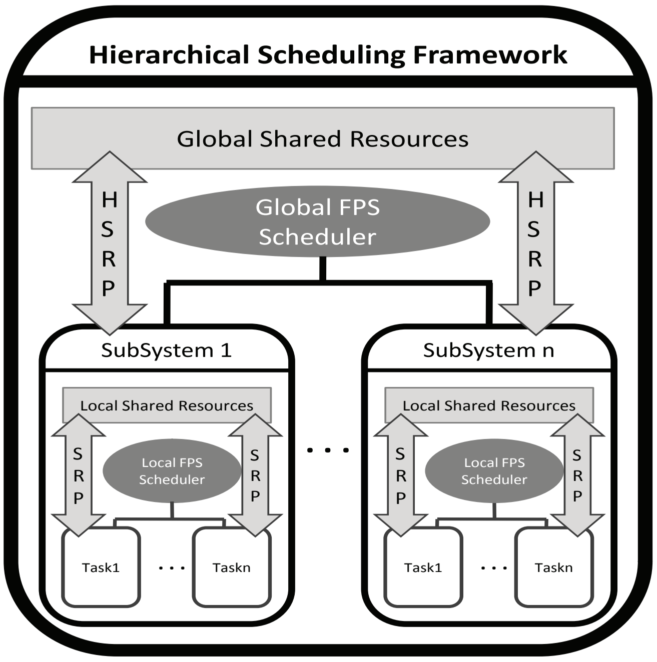


Figure 8.2: Two-level Hierarchical Scheduling Framework

The HSF gives the potential to develop and analyze subsystems in isolation from each other [26]. As each subsystem has its own local scheduler, after satisfying the temporal constraints, the temporal properties are saved within each subsystem. Later, a global scheduler is used to schedule all the subsystems together without violating the temporal constraints that are already analyzed and stored in the subsystems. Accordingly we can say that the HSF provides partitioning of the CPU between different servers. Thus, server-functionality can be isolated from each other for, e.g., fault containment, compositional verification, validation and certification, and unit testing.

Using HSF a subsystem (runnable virtual node in our case) can be developed and analyzed in isolation, with its own local scheduler at first step of deployment and its temporal properties are preserved. Then at the second step of deployment, multiple virtual nodes (subsystems) are integrated onto a physical node using an arbitrary global scheduler without violating the temporal properties of the individual subsystems analyzed in isolation.

8.3.3 FreeRTOS and its HSF Implementation

In this work we use FreeRTOS as the operating system to execute both levels of the HSF. FreeRTOS is a portable open source real-time kernel with properties like small and scalable, support for 23 different hardware architectures, and ease to extend and maintain.

The official release of FreeRTOS only supports a single level fixed-priority scheduling. However, a recent work has been presented that implements a two-level HSF for FreeRTOS [7] with associated primitives for hard real-time sharing of resources both within and between servers [27]. The HSF implementation supports two kinds of servers, idling periodic [28] and deferrable servers [29]. The implementation uses fixed priority preemptive scheduling (FPPS) for both global and local-level scheduling. For local resource-sharing (within a server) the Stack Resource Policy (SRP) [30] is used, and for global resource-sharing (between servers) the Hierarchical Stack Resource Policy (HSRP) protocol [31] is used with three different overrun mechanisms to deal with the server budget expiration within the critical section [32]. The HSF supports reservations by associating a tuple $< Q, P >$ to each server where P is the server period and Q ($0 < Q \leq P$) is the allocated portion of P.

The FreeRTOS has been adopted as one of the supported operating systems of ProCom. Given Q, P, and information on resource holding times, the schedulability of a server and/or a whole system can be calculated with the methods presented in [27].

8.4 Runnable Virtual Node

The concept of runnable virtual node is used to achieve not only temporal isolation and predictable temporal properties of real-time components but also to get better reusability of components with real-time properties. Further it reduces efforts related to system-level testing, validation and certification. This concept is based on a two-level deployment process. It means that the whole system is generated in two steps rather than a single big synthesis step. At the first level of deployment, the functional properties (functionality of components) are combined and preserved with their extra-functional properties (timing requirement) in the runnable virtual nodes. In this way it encapsulates the behavior with respect to timing and resource usage, and becomes a reusable component in addition to the design-time components. Followed by the second level of deployment where these runnable virtual nodes are implemented on the physical platform along with a global scheduler.

A runnable virtual node includes the executable representation of the components assigned to the tasks, a resource allocation (period and budget of server), and a real-time scheduler, to be executed within a server in the hierarchical scheduling framework.

8.4.1 Applying Virtual Node Concept to ProCom Component Model

In ProCom, a virtual node is an integrated model concept. It means that the virtual nodes exist both on the modeling level and on the synthesis level as shown in Figure 8.1. In the synthesis realm they are called runnable virtual nodes.

At the modeling level, each virtual node contains a set of integrated ProSys components plus the execution resources (a period and budget) required for these ProSys components. The priorities of virtual nodes cannot be assigned at the modelling level. The priorities of a component are relative to other components in the system. Since virtual nodes are developed independently and are meant to be reused in different systems, therefore, the priorities are assigned to virtual nodes later during the synthesis process at the execution level.

At the execution level, the runnable virtual node contains a set of executable tasks, resources required to run those tasks and a real-time local scheduler to schedule these tasks. Note that the runnable virtual node is generated as a result of first deployment step; it is platform independent and not executable as a stand-alone entity.

8.4.2 The Synthesis of the Final Executables

The final executables are generated by assigning priorities to the servers and tasks in the runnable virtual nodes, completing the task bodies with the user code, synthesizing communication among those nodes (if needed) and linking them together with the operating system. These executables then can be downloaded and executed on a physical node. As the real-time properties of runnable virtual nodes are preserved within the servers, therefore when integrated with other runnable virtual nodes on a physical node, the real-time properties of the whole integrated system will be guaranteed by the schedulability analysis of the whole system.

The communication among runnable virtual nodes is provided by a System server, which is automatically generated for inter-node communication (if needed), at this step. The main functionality of the server is to send and receive messages among the nodes. It contains two tasks to achieve this purpose: a sender task and a message-port updater task. Additionally there can be a hardware-driver tasks in it, if needed. The system server has the highest priority of all the servers in the system, with a very small execution time and its only functionality is to copy the messages from the sender port of one component to the receiver port of another component.

An idle server is also generated within the two-level hierarchical scheduling to test the temporal isolation among the runnable virtual nodes. When there is no other server in the system to execute, then the idle server will run. It has the lowest priority of all the other servers, i.e. 0. It contains only an idle task to execute [7]. This is useful for maintaining and testing the temporal separation among servers and also useful in testing system behavior. This information is useful in detecting over-reservations of server budgets and can be used as feedback to resource management.

8.5 Case Study: Cruise controller and an adaptive cruise controller

The PROGRESS Integrated Development Environment (PRIDE) tool [33] supports development of systems using ProCom component models and it has been used for developing the examples of cruise controller (CC) and an adaptive cruise controller (ACC). The purpose of these fictitious examples is to evaluate and demonstrate the execution-time properties and reusability of the run-time components with real-time properties. First, the CC system is realized and exercised to test the temporal isolations among run-time components. Its basic functionality is to keep the vehicle at a constant speed. Then the ACC system extends this functionality by keeping a distance from the vehicle in front by autonomously adapting its speed to the speed of the preceding vehicle and by providing emergency brakes to avoid collisions. To evaluate the reusability of real-time components, the ACC system is realized by the reuse of some runnable virtual nodes from the CC system.

In the remainder of this section we describe the development of both applications using the ProCom component model. The ProSave, ProSys, and virtual node components are modeled for both examples. The presentation is followed by the synthesis of executable binaries using a hierarchical scheduling technique and the evaluation of final executables on the target platform.

8.5.1 System design

The CC system is designed from two ProSys components, Cruise Controller and Vehicle Controller, which are modelled and deployed on two different virtual nodes. For the ACC system, the Cruise Controller component is replaced with the Adaptive Cruise Controller component as shown in Figure 8.3. These virtual nodes communicate with each other through input and output message ports. The detailed design of these ProSys components is in turn shown in Figures 8.4, 8.5, and 8.6.

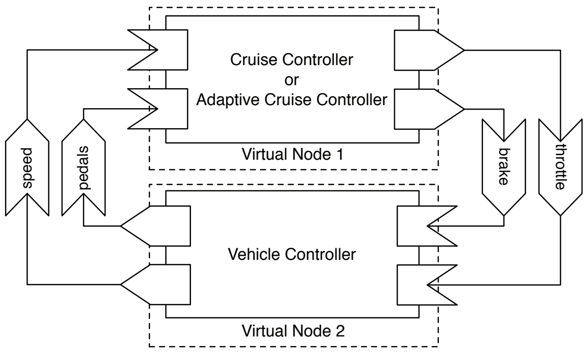


Figure 8.3: Deploying ProSys components on virtual nodes

The Cruise Controller component contains three elements as shown in Figure 8.4: an HMI Input to set the mode to on or off, and detecting the speed or the manual braking signal respectively, a Control Unit to compare the current speed with the desired speed and to send the signals to throttle or brake output port accordingly, and an HMI Output to communicate the status to the driver via the display. The Vehicle Controller component contains seven elements as shown in Figure 8.5: two Calc Max Value components: to choose the maximum (of throttle and input message port) speeds and maximum (of brake pedal and input message port) brakes, and to provide these values to Engine Controller and Brake Controller components respectively. The Speedometer writes current speed to the output port periodically.

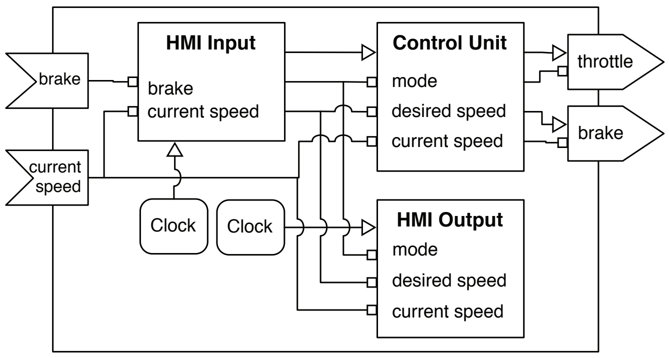


Figure 8.4: The Cruise Controller (CC) component

To extend the functionality to the ACC system, the CC virtual node is replaced by an ACC virtual node containing the same interface as that of CC component. The ACC system reuses the Vehicle Controller component from the CC application. The ACC component contains the following elements in addition to the Cruise Controller's elements: a Distance Sensor component to evaluate the distance to a vehicle/obstacle in front of the vehicle, a SpeedLimiter component to compute the vehicle's desired speed relative to the vehicle/object ahead as shown in Figure 8.6.

The ProSys components are then mapped to the virtual node components. Each virtual node is assigned a period and an execution budget to be executed in a local server within a two-level hierarchical scheduling framework. For the CC system, the Cruise Controller component is mapped to the `Virtual Node1` and the Vehicle Controller component is mapped to the `Virtual Node2`. In ACC system, the Adaptive Cruise Controller component is mapped to the `Virtual Node1` while `Virtual Node2` is reused from the CC sys-

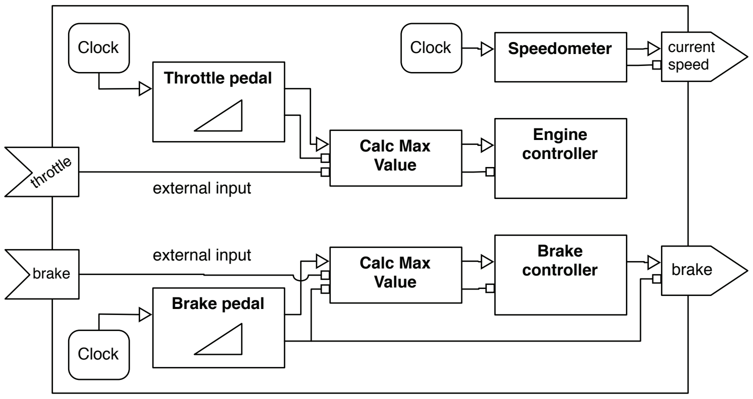


Figure 8.5: The Vehicle Controller (VC) component

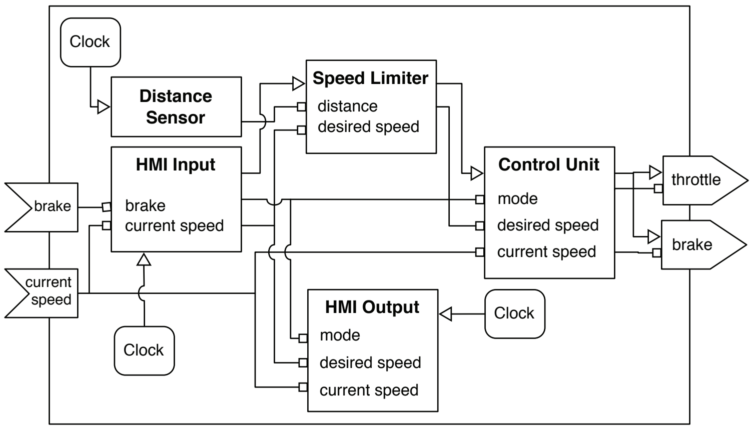
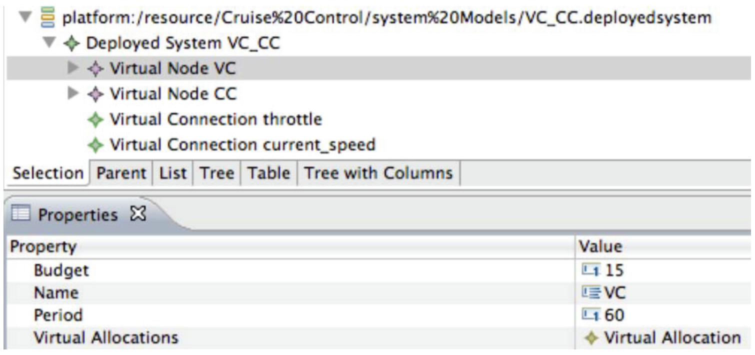


Figure 8.6: The Adaptive Cruise Controller (ACC) component

tem. The periods and budgets for these virtual nodes are assigned at the modelling level as shown in Figure 8.7.

8.5.2 Synthesis

As described in Section 8.4.2, the PRIDE tool automatically synthesizes the code from the ProCom models at different stages. It takes the models as input,



platform:/resource/Cruise%20Control/system%20Models/VC_CC.deployedsystem
Deployed System VC_CC
Virtual Node VC
Virtual Node CC
Virtual Connection throttle
Virtual Connection current_speed

Property	Value
Budget	15
Name	VC
Period	60
Virtual Allocations	Virtual Allocation

Figure 8.7: The timing properties of the virtual node

and generates all low-level platform independent code.

In the first step of the final synthesis/deployment process for the case study, two runnable virtual nodes are produced for both CC and ACC systems: one runnable virtual node for `Virtual Node1` and one for `Virtual Node2`. These generated nodes contain tasks definitions in them.

One task is synthesized for each clock. For CC example, two tasks are generated for the CC component: `CCT1` task including HMI Input and Control Unit; and `CCT2` task including HMI Output component. Three tasks are generated for the VC component: `VCT1` task including Throttle pedal, Calc Max Value, and Engine Controller; `VCT2` task including Brake pedal, Calc Max Value, and Brake Controller; and `VCT3` task including the Speedometer.

For ACC example, three tasks are generated for the ACC component: `ACCT1` task including Distance Sensor, `ACCT2` task including HMI Input, Speed Limiter, and Control Unit; and `ACCT3` task including HMI Output component.

Generating Final Binaries: In the second step of the final synthesis/deployment part, the priorities are assigned to the runnable virtual nodes (also called servers now) and to the tasks in them. Four servers are generated for both examples.

Server	CC	ACC	VC	SYSTEM
`Priority`	2	2	1	7
`Period`	40	40	60	20
`Budget`	10	10	15	4

Table 8.1: Servers used to test the CC and ACC systems behaviors.

A `System` server is generated to provide communication among the runnable virtual nodes. It has the highest priority of all the other servers, i.e. 7 (there are 8 different server priorities: from lowest priority 0 to the highest 7). Note that higher number means higher priority for both servers and tasks. The `System` server contains two tasks: a `Sender` and a `Receiver` task; whose functionality is to send and receive the data shared among virtual nodes respectively.

An `Idle` server is generated in the system with the lowest priority of all the other servers, i.e. 0, containing an idle task in it. All the other servers in the system have the priority higher than 0. This server is useful to check the temporal separation among servers.

The CC system contains two more servers in addition to `System` and `Idle` server: a `CC` server and a `VC` server associated with CC and VC virtual nodes respectively. The ACC system also contains four servers: an `ACC` server associated with the ACC virtual nodes. It reuses the `VC`, `System`, and `Idle` servers from the CC system. The priorities, periods and budgets for these servers are given in Table 8.1.

All the servers in both examples are *idling periodic* means that the tasks in the server execute and use the server's capacity until it is depleted. If server has the capacity but there is no task ready then it simply idles away its budget until a task becomes ready or the budget depletes. If a task arrives before the budget depletion, it will be served. An *idle task* per server is also generated that has the lowest priority and runs when its server has budget remaining but none of its task are ready to execute. Task properties and their assignments to the servers are given in Tables 8.2 and 8.3.

Tasks	CCT1	CCT2	ACCT1	ACCT2	ACCT3	VCT1	VCT2	VCT3
`Server`	CC	CC	ACC	ACC	ACC	VC	VC	VC
`Priority`	2	1	2	2	1	1	1	2
`Period`	40	60	40	40	60	60	60	40

Table 8.2: Tasks in the two servers.

Tasks	Sender	Receiver
`Server`	$System$	$System$
`Priority`	2	2
`Period`	20	20

Table 8.3: Tasks in the System server.

Once all the platform dependent user code is finalized, all runnable virtual nodes that are to be deployed on the same physical node are integrated with a real-time time scheduler, the platform dependent final binaries are generated and downloaded on an ECU. Currently the PRIDE tool is evolving and the automatic synthesis part is not fully mature. Hence few parts of these experiments were synthesized manually, but it is not relevant for our experiments and does not effect our results.

8.5.3 Evaluation and Discussion

We have performed the experimental evaluation of the case study on an AVR-based 32-bit `EVK1100` board [11]. The `AVR32UC3A0512` micro-controller runs at the frequency of `12MHz` and its tick interrupt handler at `1ms`(milli seconds). The FreeRTOS operating system with its HSF implementation is used on the micro-controller using idling periodic servers and FPPS scheduling policy at both levels. Its tick-handler runs at the rate of `1ms`.

Our evaluation focuses mainly to evaluate the timing properties of the real-time components during their integration and the reuse of the components in different systems. We tested the real-time components for: (i) temporal isolation among the components that leads to (ii) the predictable integration and (iii) increased reusability of the components. The experiments are described below and the results are presented in the form of visualization of servers executions in Figures 8.8, 8.9 and 8.10.

Testing temporal isolation and predictable integration

To test the temporal isolation among runnable virtual nodes and their predictable integrations, the CC system is synthesized with the previously described four servers and task sets belonging to those servers. The final binaries are executed on the micro-controller and the traces of executions are visualized. The servers executions (according to their resource reservations) along with their task sets are presented in Figure 8.8 and Figure 8.9.

In these Figures, the horizontal axis represents the execution time starting from 0. In the task's visualization, the arrow represents task arrival and a gray rectangle means task execution. In the server's visualization, the numbers along the vertical axis are the server's capacity, the diagonal line represents the server execution while the horizontal line represents either the waiting time for the next activation (when budget has depleted) or the waiting for its turn to execute (when some other server is executing). Since these are idling periodic

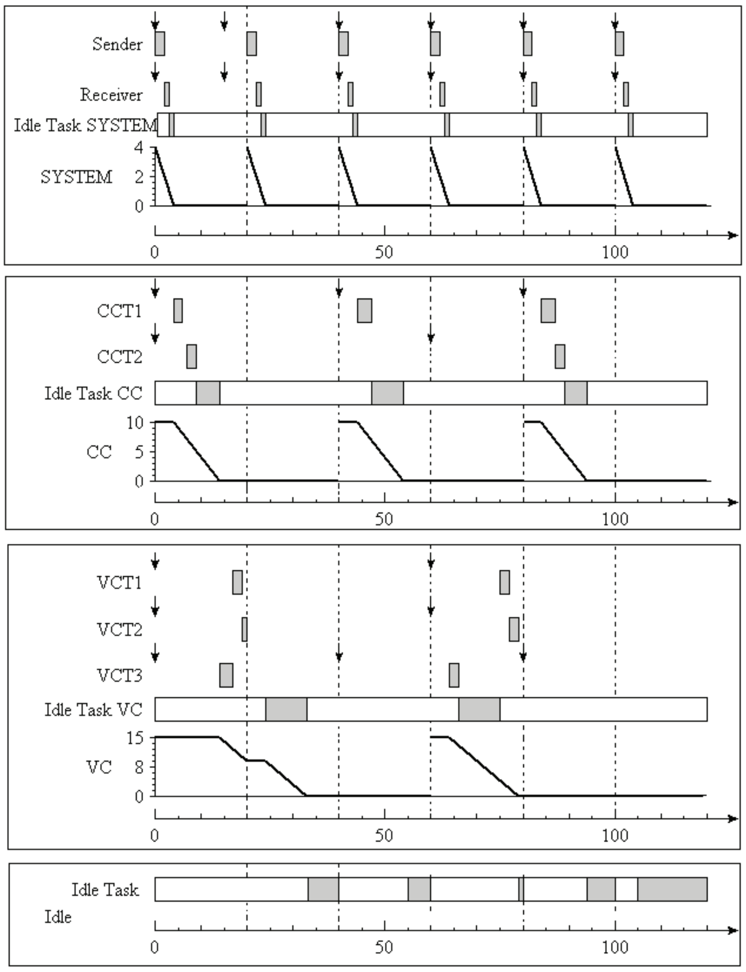


Figure 8.8: The trace for servers in the CC system during normal load

servers, all the servers in the system executes till budget depletion, if no task is ready then the idle task of that server executes till its budget depletion.

Figure 8.8 demonstrates the system execution under the normal load situation. The system's behavior is also tested during the overload situation to test the temporal isolation among the runnable virtual nodes. For example, if one server (runnable virtual node) is overloaded and its tasks miss deadlines, it should not affect the behavior of other servers in the system.

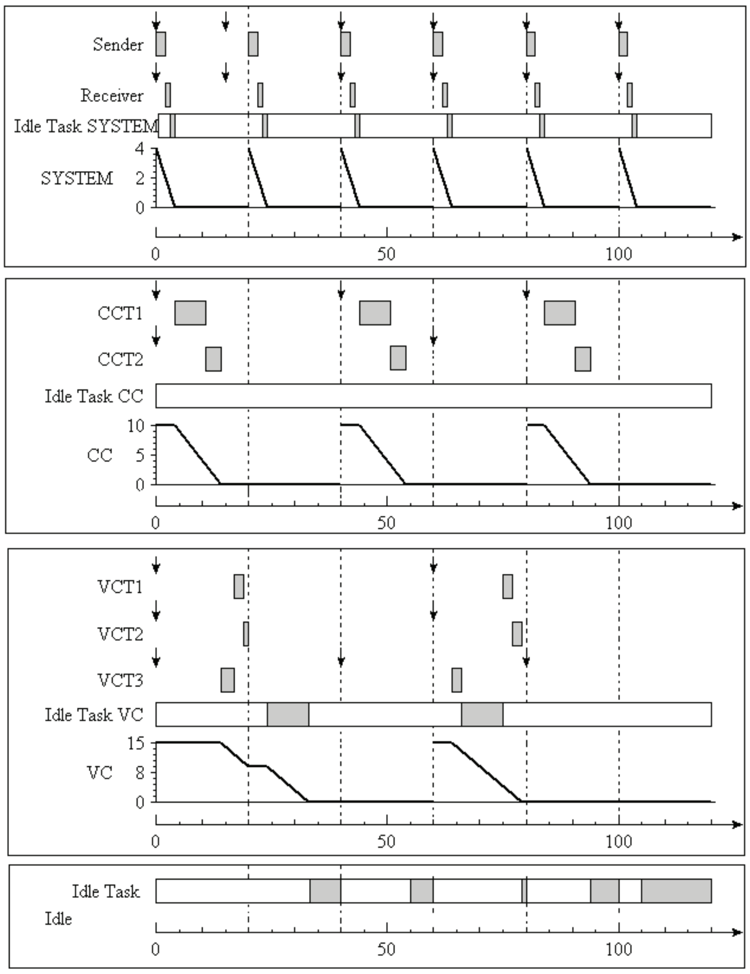


Figure 8.9: Trace showing temporal isolation during overload situation

The same example is executed to perform this test but with the increased utilization of the `CC` server as shown in Figure 8.9. The execution times of tasks `CCT1` and `CCT2` are increased by adding the busy loops, hence making the CC server's utilization greater than 1. Therefore the low priority task CCT2 misses its deadlines at time 54. CCT2 is preempted at time 14 because of the CC server's budget expiration, and starts it's execution again when next time the server is replenished. Further, the CC is never idling because it is overloaded (Idle task of CC server is not executed in Figure 8.9).

The overload of CC server does not effect the behavior of any other server in the system as obvious from Figure 8.9. The VC server has a lower priority than the CC, but still it receives its allocated resources and its tasks meet their deadlines. In this manner, the runnable virtual nodes exhibit a predictable timing behaviour that eases their integration. It also manifests that the temporal errors are contained within the faulty runnable virtual node only and their effects are not propagated to the other nodes in the system.

Testing component's reusability

The purpose of this experiment is to test the reusability of the runnable virtual nodes in a new system. The ACC system is synthesized for this purpose. It also contains four servers: the ACC server is synthesized with its task set while the other three servers are reused from the CC system. The trace of execution is visualized and presented in Figure 8.10.

Since the runnable virtual nodes preserve their timing properties within them; therefore, their behaviour should not be changed when integrated into a new system, as long as their reserved resources are provided.

The task set for the ACC server is different from that of CC server. It is clear from the Figure 8.10 that all the three reused servers sustain their timing behaviour. For example, the VC server has a lower priority than ACC, still it's behaviour is not effected at all and remains similar to its behaviour in the CC system. It confirms the predictable integration of real-time components on one hand, and demonstrates their reusability on the other hand. We observed the same results on testing the ACC server with the changed timing properties, i.e. period 40 and budget 15. As long as the allocated budgets to servers (at the modeling level) are provided, the timing properties are guaranteed at the execution.

Hence, by the use of runnable virtual node components and two-level deployment process, the timing requirements are also encapsulated within the components along with their function requirements and the temporal partitioning is provided among the components (using HSF), that results in the increased predictability during component's integration and making the runnable virtual nodes a reusable entity.

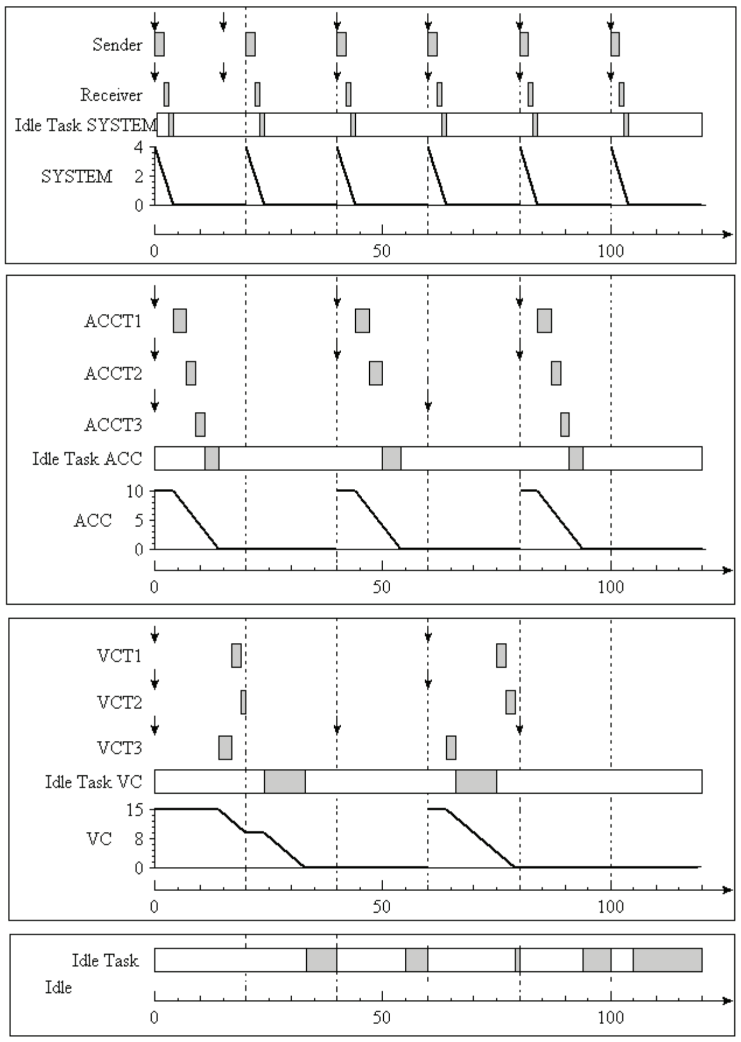


Figure 8.10: Trace showing reusability of runnable virtual nodes in ACC system

8.6 Conclusions

We present the concept of runnable virtual nodes as a means to achieve predictable integration and reuse of software components using a two-level deployment process in cyber-physical systems. The virtual node is intended as a coarse-grained component for single node deployment and with potential internal multitasking. Each physical node is used to execute one or more virtual nodes. The idea is to encapsulate real-time properties into model-driven reusable components-based systems to achieve predictable integration and reusability of virtual nodes, thereby facilitating the development of complex CPS.

The notion of two-level deployment process encapsulates the timing properties and uses the hierarchical scheduling within runnable virtual nodes that provides temporal isolation and increases the reuse of the nodes in different systems. Hence using runnable virtual nodes, a complex CPS can be developed as a set of well defined reusable components encapsulating functional and timing properties.

A proof-of-concept case study is presented which demonstrates temporal error containment within a virtual node as well as reuse of a virtual node in new environment without altering its temporal behavior. Our work is based on the ProCom component-technology [9] running on FreeRTOS which has been extended with a hierarchical scheduling framework. The case study was executed on an ECU with an AVR based 32-bit micro-controller. However, we believe that our concept is applicable also to commercial component technologies like AADL, AUTOSAR, Rubus [5].

For future work, we plan to support virtual communication-busses using server-based scheduling techniques for e.g. CAN [34] and Ethernet [35]. This will allow development, integration and reuse of distributed components using a set of virtual nodes and buses.

Bibliography

[1] Ivica Crnkovic and Magnus Larsson, editors. *Building Reliable Component-Based Software Systems*. Artech House publisher, 2002. ISBN 1-58053-327-2.

[2] L. Sha, T. Abdelzaher, K-E. rzn, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real Time Scheduling Theory: A Historical Perspective. *Real-Time Systems*, 28(2/3):101–155, 2004.

[3] J.A. Stankovic, M. Spuri, M. Di Natale, and G.C. Buttazzo. Implications of Classical Scheduling Results for Real-Time Systems. *IEEE Computer*, pages 16–25, June 1995.

[4] Jan Carlson, Juraj Feljan, Jukka Mäki-Turja, and Mikael Sjödin. Deployment Modelling and Synthesis in a Component Model for Distributed Embedded Systems. In *Proceedings of the 36th Euromicro Conference on Software Engineering and Advanced Applications (SEAA' 10)*, September 2010.

[5] Rafia Inam, Jukka Mäki-Turja, Jan Carlson, and Mikael Sjödin. Using temporal isolation to achieve predictable integration of real-time components. In *22nd Euromicro Conference on Real-Time Systems (ECRTS10) WiP Session*, pages 17–20, July 2010.

[6] Z. Deng and J.W.-S. Liu. Scheduling real-time applications in an open environment. In *IEEE Real-Time Systems Symposium(RTSS'97)*, pages 308–319, 1997.

[7] Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, S. M. H. Ashjaei, and Sara Afshar. Hierarchical Scheduling Framework Implementation in FreeRTOS. In *IEEE International Conference on Emerging Technologies and*

Factory Automation (ETFA' 11), pages 1–10, Tolouse, France, September 2011. IEEE Computer Society.

[8] Thomas Nolte. Compositionality and CPS from a Platform Perspective. In *Proceedings of the 1st International Workshop on Cyber-Physical Systems, Networks, and Applications (CPSNA'11), satellite workshop of 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'11)*, August 2011.

[9] Séverine Sentilles, Aneta Vulgarakis, Tomáš Bureš, Jan Carlson, and Ivica Crnković. A Component Model for Control-Intensive Distributed Embedded Systems. In *11th International Symposium on Component Based Software Engineering*, pages 310–317, October 2008.

[10] FreeRTOS web-site. http://www.freertos.org/.

[11] ATMEL EVK1100 product page. http://www.atmel.com/dyn/Products/tools_card.asp?tool_id=4114.

[12] AUTOSAR GbR. Specification of Operating System, 2008.

[13] Thierry Rolina. Past, Present, and Future of real-time embedded automotive software: a close look at basic concepts of AUTOSAR. In *SAE World Congress and Exhibiion, Session: In-Vehicle software*, 2006.

[14] AUTOSAR Partnership. Specification of RTE V2.0.1 R3.0 Rev 0001 , 2008. http://www.autosar.org/.

[15] Kaj Hänninen, Jukka Mäki-Turja, Mikael Nolin, Mats Lindberg, John Lundbäck, and Kurt-Lennart Lundbäck. The Rubus Component Model for Resource Constrained Real-Time Systems. In *3rd International Symposium on Industrial Embedded Systems*, 2008.

[16] Arcticus Systems Web-Page. http://www.arcticus.se.

[17] Arcticus Systems. The Rubus Operating System. http://www.arcticus.se.

[18] SAE International. AADL specification. http://www.sae.org/.

[19] G. Lasnier, B. Zalila, L. Pautet, and J. Hugues. *OCARINA : An Environment for AADL Models Analysis and Automatic Code Generation for High Integrity Applications*. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-01923-4.

[20] J. Hugues, B. Zalila, L. Pautet, and F. Kordon. From the prototype to the final embedded system using the ocarina aadl tool suite. *ACM Trans. Embed. Comput. Syst.*, 7(4):1–25, 2008.

[21] Julien Delange, L. Pautet, A. Plantec, M. Kerboeuf, F. Singhoff, and F. Kordon. Validate, simulate, and implement arinc653 systems using the aadl. *Ada Lett.*, 29(3):31–44, 2009.

[22] Object Management Group. Deployment and Configuration of Component-based Distributed Applications Specification, 2006. v4.0.

[23] Patricia Lopez Martinez and Cesar Cuevas and Jose M. Drake. RT-D&C: Deployment Specification of Real-time Component-based Applications. In *36th EUROMICRO Conference on Software Engineering an dAdvanced Applications (SEAA'10)*, pages 147–155, 2010.

[24] Tomáš Bureš, Jan Carlson, Ivica Crnković, Séverine Sentilles, and Aneta Vulgarakis. ProCom – the Progress Component Model Reference Manual, version 1.0. Technical Report MDH-MRTC-230/2008-1-SE, Mälardalen University, June 2008.

[25] Etienne Borde and Jan Carlson. Towards verified synthesis of procom, a component model for real-time embedded systems. In *14th International ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE)*. ACM, June 2011.

[26] Thomas Nolte, Insik Shin, Moris Behnam, and Mikael Sjödin. A Synchronization Protocol for Temporal Isolation of Software Components in Vehicular Systems. *IEEE Transactions on Industrial Informatics*, 5(4):375–387, November 2009.

[27] Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, and Moris Behnam. Hard Real-time Support for Hierarchical Scheduling in FreeRTOS. In *7th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT' 11)*, pages 51–60, Porto, Portugal, 2011.

[28] L. Sha, J.P. Lehoczky, and R. Rajkumar. Solutions for some Practical problems in Prioritised Preemptive Scheduling. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, pages 181–191, 1986.

[29] J.K. Strosnider, J.P. Lehoczky, and L. Sha. The deferrable server algorithm for Enhanced Aperiodic Responsiveness in Hard Real-time Environments. *IEEE Transactions on Computers*, 44(1), 1995.

[30] T. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems*, 3(1):67–99, 1991.

[31] R. I. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *IEEE Real-Time Systems Symposium (RTSS'06)*, pages 389–398, 2006.

[32] Moris Behnam, Thomas Nolte, Mikael Sjödin, and Insik Shin. Overrun Methods and Resource Holding Times for Hierarchical Scheduling of Semi-Independent Real-Time Systems. *IEEE Transactions on Industrial Informatics*, 6(1), February 2010.

[33] PRIDE Team. PRIDE: the PROGRESS Integrated Development Environment, 2010. "http://www.idt.mdh.se/pride/?id=documentation".

[34] Thomas Nolte, Mikael Nolin, and Hans Hansson. Real-Time Server-Based Communication for CAN. *IEEE Transaction on Industrial Electronics*, 1(3):192–201, April 2005. Citations=33.

[35] Rui Santos, Paulo Pedreiras, Moris Behnam, Thomas Nolte, and Luis Almeida. Hierarchical server-based traffic scheduling in ethernet switches. In *3rd Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS'10)*, pages 69–70, December 2010.