Mälardalen University Doctoral Thesis
No.169

Hierarchical scheduling for predictable execution of real-time software components and legacy systems

Rafia Inam

December 2014

MÄLARDALEN UNIVERSITY
SWEDEN

School of Innovation, Design and Engineering
Mälardalen University
Västerås, Sweden

Copyright © Rafia Inam, 2014
ISSN 1651-4238
ISBN 978-91-7485-179-3
Printed by Arkitektkopia, Västerås, Sweden
Distribution: Mälardalen University Press

Abstract

This dissertation presents techniques to achieve predictable execution of coarse-grained software components and for preservation of temporal properties of components during their integration and reuse.

The dissertation presents a novel concept runnable virtual node (RVN) which interaction with the environment is bounded both by 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 realization of RVN exploits techniques for hierarchical scheduling to achieve temporal isolation, and the principles from component-based software-engineering to achieve functional isolation. The proof-of-concept case studies executed on a micro-controller demonstrate the preserving of real-time properties within software components for predictable integration and reusability in a new environment, in both hierarchical scheduling and RVN contexts.

Further, a multi-resource server (MRS) is proposed and implemented to enable predictable execution when composing multiple real-time components on a COTS multicore platform. MRS uses resource reservation for both CPU-bandwidth and memory-bus bandwidth to bound the interferences between tasks running on the same core, as well as, between tasks running on different cores. The later could, without MRS, interfere with each other due to contention on a shared memory-bus and memory. The results indicated that MRS can be used to "encapsulate" legacy systems and to give them enough resources to fulfill their purpose. In the dissertation, the compositional schedulability analysis for MRS is also provided and an experimental study is performed to bring insight on the correlation between the server budgets.

We believe that the proposed approaches enable a faster software integration and support legacy reuse and that this work transcend the boundaries of software engineering and real-time systems.

Populärvetenskaplig Sammanfattning

Användningen av programvaror växer dag för dag i inbyggda datorsystem i tex bilar, flygplan, maskiner, och hushållsprodukter. De vanliga inbäddade programen ska ge korrekta resultat inom strikta tidsgränser och kallas realtidsprogram. Eftersom funktionaliteten och storleken hos inbyggd programvara ökar, och flera realtidsprogram integreras och distribueras på en enda hårdvaruplattform, så måste dessa programvaror dela på tillgänglig resurser. Denna delning av resurser gör det extra svårt att hinna ge resultaten inom korrekt tid, och när flera progam integreras så blir svarstiderna oftast oförutsägbara. I denna avhandling presenterar vi lösningar på problemet med förutsägbar integration av realtidsprogramvara på både traditionella (sk single-core) och moderna (sk multicore) plattformar.

För enkelkärninga plattformar, föreslår vi ett nytt koncept, en körbar virtuell nod (RVN) som kan bevara tidsegenskaper hos program när det integreras med andra program eller när återanvändas i en ny miljö. Vår realisering av RVN utnyttjar den senaste tekniken för hierarkisk schemaläggning för att uppnå tidsisolering mellan program. Fallstudier (proof- of-concept) visar bevarande av tidsegenskaper under både integration och under återanvändning i en ny miljö.

På flerkärniga plattformar, delas även andra än hårdvaruresurser mellan flera program som tex finns delat minne och delad minnesbuss. Vi utökar den hierarkiska schemaläggning för att göra den lämplig för flerkärniga plattformar genom att föreslå och genomföra en nytt koncept, som kallas Multi- Resource Server (MRS) . Den delar upp tillgång till både CPU-tid och minnesbandbredd så att varje program får sina nödvändiga resurser. MRS ger tidsisolering mellan program som körs på samma kärna, liksom mellan program som körs på olika kärnor. De senare skulle, utan MRS, störa varandra på grund av begränsad åtkomst till det delade minne. Vi visar i en fallstudie (proof-of-concept) att MRS är också lämplig för att migrera gamla befintliga program till multicores.

To the

dream of my parents,

guidance of my supervisors,

patience of my husband and

endless love of my daughters.

Acknowledgements

Few years ago, PhD was a big goal for me to achieve. During all those years, I learned that the whole joy is not only reaching the final destination, but also the journey towards the PhD title.

There are many people who have come along with me in this journey and helped me to turn it into an achievement. The most important figures are of course my supervisors Professor Mikael Sjödin, Dr. Jukka Mäki-Turja and Dr. Moris Behnam. First of all, I would like to express my deepest gratitude to my main supervisor Professor Mikael Sjödin, for believing in me and giving me the opportunity to become a PhD student. The work presented in this dissertation would not have been possible without his expert guidance, invaluable inputs, support and encouragement, and always finding time to help me. I am grateful to my assistant supervisors for providing valuable and useful suggestions, comments and feedback throughout my studies. I had a great opportunity of learning so many new things from them during our meetings and discussions.

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

I thank Dr. Jan Carlson and Dr. Wasif Afzal for providing valuable comments on the dissertation.

I attended several courses during my PhD studies. I thank the lecturers and professors including Hans Hansson, Ivica Crnkovic, Thomas Nolte, Emma Nehrenheim, Mikael Sjödin, Jukka Mäki-Turja, Daniel Sundmark, Lena Dafgård for their guidance during my studies. I want to also thank all other faculty members, they have been a source of inspiration for me.

I want to thank Farhang, Notle, Jan, Jiří, Matthias, Marcus, and Daniel Cederman - whom I have enjoyed working with. Special thanks to Reinder

J. Bril for providing a positive feedback on the work that we did together. I supervised multiple master students during PhD. I wish all of them best of luck.

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 gratitude to Hans Hansson and my supervisors for helping me to find new job.

I would also like to thank my friends and colleagues at the department for all the fun we had during my studies, conference trips, coffee breaks and parties. I wish to thank Sara Ab., Farhang, Sara D., Adnan, Danial, Wasif, Eduard, Raluca, Guillermo Sara Ab., Alessio, Kan, Leo, Nesredin, Predrag, Dag, Severine, Mohammad, Saad, Svetlana, Elena, Hamid, Matthias, Meng, Sara Af., Nima, Simin, Cristina, Moris, Hang, Mobyen, Shahina, Giacomo, Anna, Ning, Andreas G., Lars, Mattias, Bob, Batu, Fredrik, Nikola, Abhilash, Andreas J., Kaj, Zhou, Hseyin, Patrick, Hongyu, Gaetana, Barbara, Radu, Sasi, Antonio, Federico, Mehrdad, Gabriel, Irfan, Mahnaz, Omar, Ana P., Josip, Juraj, Luka, Adam, Aida, Andreas H., Aneta, Frank, Jagadish, Jiri, Kivanc, Mikael Å, Stefan B., Thomas L., Tibi and others for all the fun and memories.

Finally, I would like to extend my deepest gratitude to my family. Many thanks go to my parents for filling my head with dreams, for their love, support and encouragement. My deepest gratitude goes to my husband Inam for encouraging me to fulfill the dreams, for being always with me, with a positive and supportive attitude, in all these rough and tough days, and to my daughters Youmna and Urwa for bringing endless love and happiness to our lives ☺.

The work in this dissertation has been supported by the Swedish Knowledge Foundation (KKS) within the project PPMSched and the Swedish Foundation for Strategic Research (SSF) with the PROGRESS.

Rafia Inam
Västerås, December, 2014

List of publications

Papers included in the PhD dissertation

Available at http://www.es.mdh.se/staff/279-Rafia_Inam[1]
See Chapter 5 for an account of my personal contribution to each paper.

Paper A *Support for Hierarchical Scheduling in FreeRTOS*. Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, S. M. H. Ashjaei, Sara Afshar. In the 16th IEEE conference on Emerging Technologies and Factory Automation (ETFA), IEEE, September, 2011.
IEEE Industrial Electronics Society Scholarship Award.

Paper B *Support for Legacy Real-Time Applications in an HSF-Enabled FreeRTOS*. Rafia Inam, Moris Behnam, Mikael Sjödin. MRTC report ISSN 1404-3041, ISRN MDH-MRTC-295/2014-1-SE, Mälardalen Real-Time Research Center, Mälardalen University, November, 2014. (submitted to journal).

Paper C *Predictable Integration and Reuse of Executable Real-time Components*. Rafia Inam, Jan Carlson, Mikael Sjödin, Jiří Kunčar. In Journal of Systems and Software (JSS), Volume 91, Number 0, Elsevier, May 2014.

Paper D *The Multi-Resource Server for Predictable Execution on Multi-core Platforms*. Rafia Inam, Nesredin Mahmud, Moris Behnam, Thomas Nolte, Mikael Sjödin. In the 20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), IEEE, April, 2014.

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

Paper E *Worst Case Delay Analysis of a DRAM Memory Request for COTS Multicore Architectures*. Rafia Inam, Moris Behnam, Mikael Sjödin. In Seventh Swedish Workshop on Multicore Computing (MCC 14), Lund, Sweden, November, 2014.

Paper F *Compositional Analysis for the Multi-Resource Server - a technical report*. Rafia Inam, Moris Behnam, Thomas Nolte, Mikael Sjödin. MRTC report ISSN 1404-3041, ISRN MDH-MRTC-283/2014-1-SE, Mälardalen Real-Time Research Center, Mälardalen University, October, 2014. (submitted to conference).

Other relevant publications

Available at http://www.es.mdh.se/staff/279-Rafia_Inam, and
http://scholar.google.com/citations?user=3hr4-M0AAAAJ&hl=sv

Licentiate Thesis

1. *Towards a Predictable Component-Based Run-Time System*. Rafia Inam. Licentiate Thesis, ISSN 1651-9256, ISBN 978-91-7485-054-3, January, 2012.

Journal publications

2. *Multi-core Composability in the Face of Memory Bus Contention*. Moris Behnam, Rafia Inam, Thomas Nolte, Mikael Sjödin. ACM SIGBED Review, Special Issue on 5th Workshop on Compositional Theory and Technology for Real-Time Embedded Systems, Vol 10, nr 3, October, 2013.

3. *Implementing and Evaluating Communication- Strategies in the ProCom Component Technology*. Rafia Inam, Mikael Sjödin. ACM SIGBED Review: Special Issue on the 24th Euromicro Conference on Real-Time Systems (ECRTS'12), vol 9, nr 4, pages 17-20, ACM, ISSN: 1551-3688, November, 2012.

4. *Virtual Node - To Achieve Temporal Isolation and Predictable Integration of Real-Time Components*. Rafia Inam, Jukka Mäki-Turja, Jan Carlson, Mikael Sjödin. Journal on Computing, vol 1, nr 4, GSTF Publishing, January, 2012.

Conference and Workshop publications

5. *Towards improved dynamic reallocation of real-time workloads for thermal management on many-cores*. Rafia Inam, Matthias Becker, Moris Behnam, Thomas Nolte, Mikael Sjödin. IEEE Real-Time Systems Symposium, (WiP) session (RTSS'14), Rome, Italy, December, 2014.

6. *Formalization and Verification of Mode Changes in Hierarchical Scheduling*. Yin Hang, Rafia Inam, Reinder J. Bril, Mikael Sjödin. 26th Nordic Workshop on Programming Theory (NWPT'14), Halmstad, Sweden, October 2014.

7. *Combating Unpredictability in Multicores through the Multi-Resource Server*. Rafia Inam, Mikael Sjödin. 2nd International Workshop on Virtualization for Real-Time Embedded Systems (VtRES'14), Barcelona, Spain, September 2014.

8. *Performance Preservation using Servers for Predictable Execution and Integration*. Rafia Inam. 38th Annual International Computers, Software and Applications Conference (COMPSAC'14), Västerås, Sweden, July 2014.

9. *Mode-Change Mechanisms support for Hierarchical FreeRTOS Implementation*. Rafia Inam, Mikael Sjödin, Reinder J. Bril. 18th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'13), September, 2013.

10. *Towards Implementing Multi-resource Server on Multi-core Linux Platform*. Rafia Inam, Joris Slatman, Moris Behnam, Mikael Sjödin, Thomas Nolte. 18th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'13), September, 2013.

11. *Real-Time Component Integration using Runnable Virtual Nodes*. Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, Jiří Kunčar. 38th Euromicro Conference on Software Engineering and Advanced Applications (SEAA' 12), Izmir, Turkey, September, 2012.

12. *Bandwidth Measurement using Performance Counters for Predictable Multicore Software*. Rafia Inam, Mikael Sjödin, Marcus Jägemar. 17th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'12), WiP, September, 2012.

13. *Implemnting Hierarchical Scheduling to support Multi-Mode System*. Rafia Inam, Mikael Sjödin, Reinder. J. Bril. 7th IEEE International Symposium on Industrial Embedded Systems (SIES'12), WiP, Karlsruhe, Germany, June, 2012.

14. *Towards Resource Sharing by Message Passing among Real-Time Components on Multi-cores*. Farhang Nemati, Rafia Inam, Thomas Nolte, Mikael Sjödin. In 16^{th} IEEE International Conference on Emerging Technology and Factory Automation (ETFA'11), Work-in-Progress session, Toulouse, France, September, 2011.

15. *Hard Real-time Support for Hierarchical Scheduling in FreeRTOS*. Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, Moris Behnam. 7th annual workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'11), p 51-60, Porto, Portugal, July, 2011.

16. *Using Temporal Isolation to Achieve Predictable Integration of Real-Time Components*. Rafia Inam, Jukka Mäki-Turja, Jan Carlson, Mikael Sjödin. In 22^{nd} Euromicro Conference on Real-Time Systems (ECRTS' 10) WiP Session, Pages 17-20, Brussels, Belgium, July, 2010.

17. *A* Algorithm for Graphics Processors*. Rafia Inam, Daniel Cederman, Philippas Tsigas. In 3^{rd} Swedish Workshop on Multi-core Computing (MCC'10), Gothenburg, Sweden, 2010.

Technical reports

18. *Worst Case Delay Analysis of a DRAM Memory Request for COTS Multicore Architectures - A technical report*. Rafia Inam, Moris Behnam, Mikael Sjödin. Technical Report, ISSN 1404-3041, Mälardalen University, October, 2014.

19. *Formalization and verification of mode changes in hierarchical scheduling - An extended report*. Hang Yin, Rafia Inam, Reinder J. Bril, Mikael Sjödin. Technical Report, ISSN 1404-3041, Mälardalen University, Sep, 2014.

20. *End-to-End Latency Analyzer for ProCom - EELAP*. Jiří Kunčar, Rafia Inam, Mikael Sjödin. Technical Report, ISSN 1404-3041, ISRN MDH-MRTC-272/2013-1-SE, Mälardalen University, March, 2013.

21. *Run-Time Component Integration and Reuse in Cyber-Physical Systems*. Rafia Inam, Jukka Mki-Turja, Mikael Sjdin, Jiří Kunčar. Technical Report, ISSN 1404-3041, Mälardalen University, December, 2011.

22. *Hierarchical Scheduling Framework Implementation in FreeRTOS*. Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, Seyed Mohammad Hossein Ashjaei, Sara Afshar. Technical Report, ISSN 1404-3041, Mälardalen Real-Time Research Center, Mälardalen University, April, 2011.

23. *An Introduction to GPGPU Programming - CUDA Architecture*. Rafia Inam. Technical Report, Mälardalen University, December, 2010.

24. *Different Approaches used in Software Product Families*. Rafia Inam. Technical Report, Mälardalen Real-Time Research Center, Mälardalen University, 2010.

Contents

I

Thesis

Chapter 1

Introduction

In embedded real-time systems, a clear trend in increasing size and complexity of embedded software has been observed during the last decades [1]. For example, a modern car contains thousands of software functions corresponding to nearly 100 million lines of code running on around 70 to 100 embedded processors [2]. To battle this trend, modern software-development technologies are being adapted 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. In this dissertation, a component is defined as a consistent set of concurrent time-constrained tasks each performing a specific functionality. These individual components are typically first developed and tested in isolation, and later integrated to create a complete software for the system [1]. Integration of real-time components can be explained as the mechanism of wiring components together [1]. Thus *integration of real-time components* is one approach to address the main challenges of increased development cost, time to market, and increased complexity of software.

Reuse of legacy code is another approach to meet these challenges. Many industrial systems are developed in an evolutionary fashion, reusing components from previous versions or from related products [1]. For example, the new Boeing 787 ”Dreamliner” is a recent example with a significant proportion of reused code from another Boeing airplane [3, 4]. Further, the advent of low cost and high performance hardware platforms have made it possible to integrate multiple complex real-time components on a single hardware node. It means that software components are reused and re-integrated in new environments.

For real-time systems, the timing requirements should be guaranteed during integration and reuse of the components which creates new challenges. The focus of this dissertation is on the *schedulability of tasks*, i.e., meeting their deadlines, as the main timing requirement. Predictability refers to the possibility to guarantee presence or absence of certain properties. Thus, a component's timing behaviour is called *predictable* during its integration and reuse, as long as the schedulability of tasks that have been validated during its development phase is guaranteed when multiple components are integrated together.

The aim of this dissertation is to investigate and propose techniques for predictable integration and execution of software components with real-time requirements. This dissertation also examines the impact of shared hardware resources of multicore platforms on the predictability of real-time software and presents solutions for it.

1.1 Proposal

In this dissertation, we propose a *Runnable Virtual Node (RVN)* concept, which is an execution-platform component that preserves functional as well as temporal properties of the software executed within it [5]. It is intended for coarse-grained components for single node deployment and with potential internal multitasking. In order to realize this concept, first a Hierarchical Scheduling Framework (HSF) technique [6, 7] is implemented and tested on a target platform (a microcontroller in our case), and later it is embedded within the RVN component. HSF is a known technique in real-time community that partitions the CPU resource among components and provides temporal isolation among the partitions. Thereby, RVN exploits the benefits of both, CBSE and HSF. It achieves advantages of encapsulating the temporal properties of real-time components and components' reusability from CBSE [1]; and predictable integration of components by rendering temporal partitioning among components [8] and independent development and analysis of components from hierarchical scheduling [7, 9, 10] approaches.

On multicore hardware platform, predictability could not be attained by simply partitioning the CPU resource due to multiple shared hardware resources like caches, memory-bus and memory. Thus, the traditional hierarchical scheduling with only CPU partitioning is not enough to use [11]. In this dissertation, we also propose a novel server-based technique called *Multi-Resource Server (MRS)* that partitions two resources, CPU and memory-bus bandwidth, in order to support more predictable execution and reuse of real-time components on multicore platform. For hard real-time systems, we also present schedulability analysis of MRS.

1.2 Background

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

1.2.1 Hierarchical scheduling framework

A hierarchical scheduling framework [6, 7] can be considered as a system consisting of a set of components or subsystems (executed as servers). A two-level HSF can be viewed as a tree with one parent node and many leaf nodes as illustrated in Figure 1.1. The parent node is a global scheduler and leaf nodes are subsystems. Each subsystem consists of its own internal set of tasks that are scheduled by a local scheduler. The global scheduler schedules the system and is responsible for dispatching the servers according to their allocated resource reservations. The local scheduler then schedules its task set according to its internal scheduling policy.

HSF supports CPU time sharing among components and isolates components' functionality from each other e.g., temporal fault containment, compositional verification, and unit testing. Further 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. Thus, using HSF, components can be developed and analyzed in isolation from each other.

In the two-level HSF, the mutually exclusive 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 inter-subsystem) called global shared resources as shown in Figure 1.1. Different synchronization protocols are required to share resources at local and global levels, for example, Stack Resource Policy (SRP) [12] can be used at local level, and at global level, Hierarchical Stack Resource Policy (HSRP) [13] can be used.

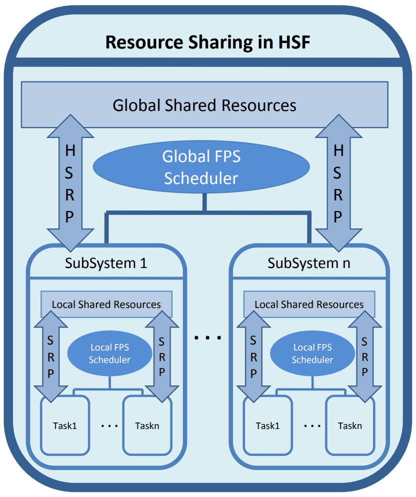


Figure 1.1: Two-level Hierarchical Scheduling Framework using Resource Sharing.

Throughout the dissertation, the terms application, subsystem or server refer to a real-time component or a component-based real-time embedded application. Further, the terms HSF and server-based scheduling are used interchangeably.

1.2.2 ProCom component model

Component-based software engineering and Model-Based Engineering (MBE) are two emerging approaches to develop embedded control systems, e.g., 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) [14].

The ProCom component model can be described in two distinct realms: the modeling and the runnable realms as shown in Figure 1.2. In Modeling

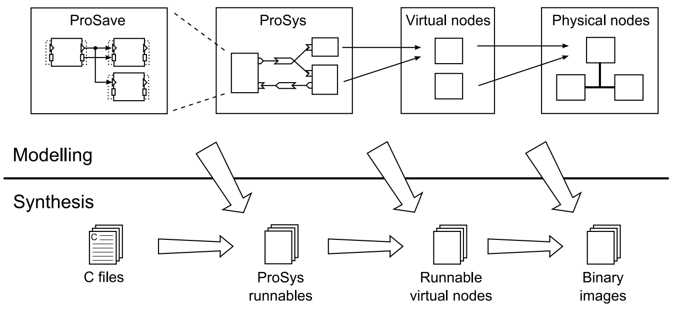


Figure 1.2: An overview of the deployment modelling formalisms and synthesis artefacts.

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 1.2. 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 [15, 16].

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 [14].

The runnable realm

At the runnable realm, runnable objects 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. Runnable virtual nodes 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.

The PROGRESS Integrated Development Environment (PRIDE) tool [17] supports the automatic synthesis of the components at different levels [18]. 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. More details on RVN can be found in [19, 5, 20] and in Section 8.

1.3 Dissertation outline

The dissertation is organized in two distinctive parts. Part-I gives a summary of the performed research. Chapter 1 presents a brief introduction and describes background of the research. Chapter 2 describes the research problem, the main research goal and research challenges used as guidelines to perform the research, and introduces the research method we used. Chapter 3 describes our technical contributions and recapitulates the research challenges. Chapter 4 concludes the thesis by summarizing the contributions and outlining the future work. Finally, Chapter 5 provides a short description and authors' contributions of the papers included in this dissertation.

Part-II includes four peer-reviewed scientific papers and two technical reports contributing to the research results. The papers are published and presented in international conferences, workshops and international journals. Both technical reports are based on peer-reviewed and published papers, and are now extended for journal/conference publishing (see Chapters 7 and 11 for details). The papers are presented in Chapters 6, 8 - 10.

Chapter 2

Research overview

2.1 Research problem and challenges

Integration of real-time applications (also referred to as components [7]) can be explained as the mechanical task of wiring components together [1]. For real-time embedded systems, the components and components integration must satisfy (1) functional requirements and (2) extra-functional requirements (e.g. satisfying timing requirements). Temporal behavior of real-time components poses big challenges in their integration. When multiple components are deployed on the same hardware node, the timing behavior of each one of them is typically altered in unpredictable ways. This means that a component that is found correct during its 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. Further the reuse of a component is restricted because it is very difficult to know beforehand if the component will be schedulable in a new system. For real-time embedded systems, methodologies and techniques are required to provide temporal isolation so that the timing requirements could be guaranteed during components' integration and reuse.

2.1.1 Predictable execution, integration and reuse of real-time components

One promising technique for integrating complex real-time components on a unicore processor to overcome above mentioned deficiencies is the hierarchical scheduling framework in which the components are executed as servers with specified budgets and periods [6, 7]. It partitions CPU time among components. It supplies an efficient mechanism (i) to provide predictable integration of components by rendering temporal partitioning among components [8], (ii) to support independent development and analysis of real-time components [7], and (iii) to provide the analysis of integrated components at the system level [7]. HSF has been proposed to develop complex real-time systems by enabling temporal isolation and predictable integration of software-functions [21].

However, integrating HSF within a component-based software technology for embedded real-time systems raises challenges of preserving the timing properties within software components to use these properties during components' integration and execution on unicore hardware platform. It includes the challenges of enriching a component model with hierarchical scheduling, i.e., allocating timing resources and a real-time scheduler to a software component and executing the component as server in the HSF. Another challenge during component integration is to provide reliable communication among various components of a target system. This communication should also be predictable in case of real-time components and should not affect the schedulability of tasks.

2.1.2 Predictability on COTS multicore platforms

Using Commercial-Off-The-Shelf (COTS) multicore platforms for real-time applications presents more challenges. One such challenge is to achieve and maintain predictable execution of concurrent tasks that compete for both CPU- and memory-bus bandwidth resources. On unicore platforms, the server-based scheduling approach successfully bounds the interference between the applications running on the same core [22, 23, 7]. However, this approach is limited in provisioning of the CPU resource only and does not take care of the activities that are located on different cores and can still interfere with others in an unpredictable manner. In multi-core platforms, concurrent tasks allocated to the same core interfere with each other by competing for CPU-bandwidth (we call this *local interference*), and concurrent tasks allocated to different cores interfere by competing for memory-bus bandwidth and shared memory (we call this *global interference*). A main source of such an unpredictable negative impact is the contention for shared physical resources like cache, memory-bus and memory.

In existing commercial hardware, there are currently no mechanisms that allow a core to protect itself from global interference if another core starts stealing its memory bandwidth or pollutes its cache lines. For performance-critical real-time systems, overcoming these problem is paramount. Thus the memory-bus bandwidth should also be considered to guarantee predictable performance of real-time applications that are located in different cores for multicore platforms, especially when migrating/reusing software from a unicore to a multicore architecture. Hence, there is a need to develop software technologies to track, and eventually police, the consumed memory-bandwidth in order to increase predictable software execution on a multi-core platform.

While the incorporation of memory-bus resource into server-based scheduling increases the predictable execution of real-time applications on different cores, the shared memory still hinders the predictability and is another challenge that we have addressed in this dissertation. The scheduling alone (i.e., controlling the allocation of resources over time) is not enough to achieve complete timing isolation. Dynamic RAM (DRAM) is another resource shared among all cores, and can have a tangible effect on timing requirements of tasks executing in different servers in different cores. There is a strong need to investigate about implementing some DRAM-partitioning techniques (like in [24]) or bounding DRAM accesses using some static analysis technique (like in [25, 26]) for COTS multicore platforms.

Further, the analysis for the memory-aware server-based scheduling approach is another challenge that is targeted in this dissertation.

2.2 Research methodology

This research work has been carried out following the deductive method [27]. The main activities are as follows:

1. Identification of the research problem from current trends from real-time and component-based communities and definition of the research goal. This activity included study of the current state-of-the-art literature, attending courses/seminars related to the topic, and inspirations from industry.

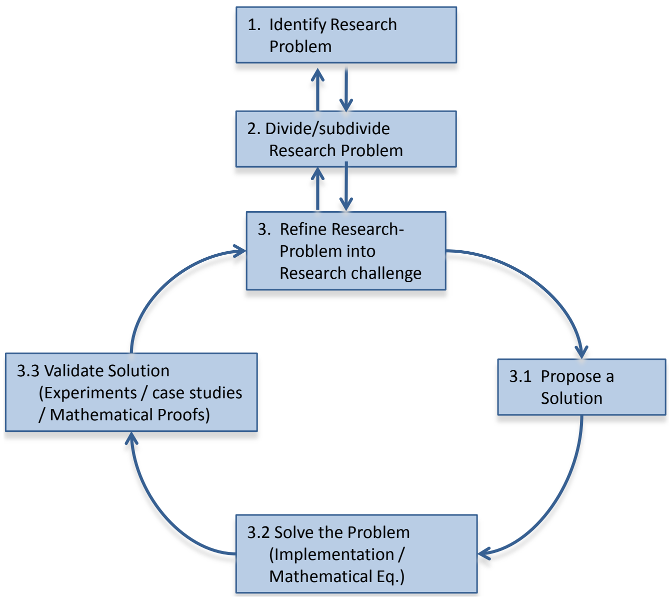

1. Identify Research Problem
2. Divide/subdivide Research Problem
3. Refine Research-Problem into Research challenge
3.1 Propose a Solution
3.2 Solve the Problem (Implementation / Mathematical Eq.)
3.3 Validate Solution (Experiments / case studies / Mathematical Proofs)

Figure 2.1: Deductive Method of Research.

2. Dividing or sub-dividing the research problem into smaller and easily manageable problems.
3. Refining the smaller problem to a research setting and defining further research guiding challenges.
    3.1 Study of the current state-of-the-art based on guiding challenges and proposing a solution.
    3.2 Solving the problem (by implementing a prototype and/or by providing mathematical proofs/analysis) and presenting the research results.
    3.3 Illustration and validation of the research results. This is done by performing case studies and/or synthetic experimental evaluations on the implementation,.

In this work, a single overarching goal is identified at step 1, which is divided into smaller research challenges, and solved one at a time using steps in 3. These steps (3.1 - 3.3) are performed for each of the guiding challenges unless the desired results for the overall goal are achieved, as described in Figure 2.1. The research performed to meet the guiding challenges of step 3 resulted in one or more research publications.

2.3 Research goal and refined research challenges

The main goal of this dissertation is:

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

The research goal is refined and formulated into the following research challenges which we have used as a guideline for our research:

- **C1** Achieving predictability during real-time components' integration and reuse. It is further subdivided into two challenges:
  - **C1.1** Developing runtime mechanisms to preserve the temporal requirements within real-time components.
  - **C1.2** Developing runtime mechanisms to facilitate legacy code migration in new environment.
- **C2** Integrating hierarchical scheduling approach within CBSE to improve today's embedded system development by reusing preserved temporal properties.
- **C3** Adapting traditional server-based scheduling to be suitable to multicore platforms.
- **C4** Providing an analysis framework for the new server-based scheduling from C3.

By addressing these challenges, we aim to develop approaches that enable predictable software integration and support legacy reuse.

Chapter 3

Technical contributions

3.1 Contributions

The main contributions of the dissertation are as follows:

Contribution 1. Implementation of HSF and legacy server for unicore platform
This contribution addresses challenges C1.1 and C1.2. We provide a two-level hierarchical scheduling support for FreeRTOS operating system with the consideration of minimal modifications in FreeRTOS kernel [28]. We extend FreeRTOS scheduler for idling periodic [29] and deferrable [30] servers using fixed-priority preemptive scheduling at both global and local levels. Further, to support resource sharing among arbitrary tasks that execute in arbitrary components, we implement Stack Resource Policy (SRP) [12] and Hierarchical Stack Resource Policy (HSRP) [13].

To address challenge C1.2 of migrating legacy systems in an HSF-setup, HSF-implementation needs adaptations to fit with legacy systems requirements. The examples of such are adapting the original OS API with the HSF-API, and the sharing both hardware and software resources of the system among legacy and other components. We enhance the implementation with a legacy server which executes the legacy code by doing minimum modifications in it. Simply using synchronization protocols at both levels of HSF does not resolve the resource sharing problem, when a legacy server is executed, which still calls old operating system API for resource sharing. One strategy is to change the legacy code with the newly developed synchronization API for HSF. However, this is tedious, time consuming, and error prone. We provide wrappers for the old API as an alternative strategy. The advantage of wrapping over conventional redevelopment is the low cost. Additionally it keeps the semantics of the original operating system code unchanged. This contribution is presented in Papers A and B.

Contribution 2. The RVN concept

To address challenges C1 and C2 in CBSE context, we present the concept of a Runnable Virtual Node (RVN) by integrating HSF into component technology to provide temporal isolations among components, which eventually leads to predictable integration of components. An RVN exhibits the functionality of a software-component (or a set of integrated components) combined with allocated timing resources and a local real-time scheduler, thus the RVN executes as a server in the HSF. 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. 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 virtual nodes. In this way it encapsulates the behaviour with respect to timing and resource usage and becomes a reusable executable component in addition to the design-time components. It is followed by the second level of deployment where multiple virtual nodes are deployed on the physical node (target hardware) along with a global scheduler.

In addition, we implement a server-based communication strategy which executes communication code in a separate server. This strategy incorporates the maintainability and flexibility to change the communication code without affecting the timing requirements of RVNs. We have evaluated the end-to-end delay analysis for the server-based strategy with a more direct communication strategy for efficiency and reusability properties of RVNs. Hence using RVNs and server-based inter-RVN communication, complex real-time systems can be developed as a set of well defined reusable components encapsulating functional and timing properties.

The work is based on the ProCom component-technology [16] running on the HSF implementation on FreeRTOS [31]. However, we believe that our concept is also applicable to commercial component technologies like AADL and AUTOSAR [5]. This contribution is presented in Paper C.

Contribution 3. Presentation and realization of MRS
This contribution addresses challenges C1 and C3, and targets statically partitioned multi-core real-time systems. For these systems we present the Multi-Resource Server (MRS) technology that schedules the resources: CPU and memory-bus bandwidth. In statically partitioned multi-core systems, concurrent tasks allocated to the same core suffer from local interference while concurrent tasks allocated to different cores interfere due to global interference of the shared hardware resources. A first step is to estimate the amount of consumed memory-bandwidth for each application. Such estimates can then be used to track down bottlenecks, provide better partitioning among cores, and ultimately be used to arbitrate and police accesses to memory-bus.

MRS enables more predictable execution of real-time applications on multi-core platforms through resource reservation approaches in the context of CPU bandwidth reservation and memory-bandwidth reservation. The MRS provides temporal isolation both between tasks running on the same core, as well as, between tasks running on different cores. The latter could, without MRS, interfere with each other due to contention on a shared memory bus.

We implement MRS as a user-space library for Linux running on COTS multicore hardware. We demonstrate that MRS can be used to preserve the functionality of a legacy application when it is executed on a single core while another core executes tasks with adverse memory behavior. We demonstrate for a synthetic task-set, how the MRS can be used to isolate tasks from each other, to prevent adverse behavior of some tasks to negatively impact other tasks. Paper D describes this contribution in detail.

Contribution 4. Case studies for proof-of-concept
In this contribution we validate our solutions of challenges C1, C2, and C3. Although synthetic experiments confirm the viability of the approaches, case studies validate the practicality of the approaches.

We validate our HSF implementation for legacy server to demonstrate the ease of using legacy server and wrappers. We execute two legacy FreeRTOS applications as an example case study which are originally developed and executed as stand-alone application for the FreeRTOS operating system on a micro-controller using EVK1100 board. We test these applications in a two-level hierarchical setup along with other applications. Our motivating case study is simple, but exercises the execution-time properties and evaluates the creation, behaviour, viability of the legacy server and the legacy tasks. It also evaluates the behaviour of wrappers for FreeRTOS API, resource sharing among legacy tasks and tasks of the new applications, and finally the behaviour of HSRP protocols. This part of the contribution is presented in Paper B.

As virtual node is an integrated concept within the ProCom component model [16], we implemented an example case study in the PRIDE tool [17] that supports the development of systems using ProCom components running on the HSF implementation on FreeRTOS [31]. We use ProCom components for development of a cruise controller (CC) and an adaptive cruise controller (ACC) for automotive applications. The proof-of-concept case study demonstrates the temporal-fault containment within an RVN as well as the reuse of RVNs in new environment thereby facilitating predictable integration. We also evaluate the end-to-end delay analysis for the server-based strategy with a more direct communication strategy for efficiency and reusability properties of RVNs. We have developed an analysis tool *End-to-End Latency Analyzer for ProCom (EELAP)* [32, 33] to automate the computations of worst case response times of tasks and calculations of different end-to-end latency semantics for multi-rate server-based ProCom components. This part of the contribution is presented in Paper C.

For MRS, we demonstrate that the MRS can be used to "encapsulate" legacy systems and to give them enough resources to fulfill their purpose in Paper D. In our case study a legacy media-player is integrated with several resource-hungry tasks running on a different core. We show that without MRS the media-player starts to drop frames due to the interference from other tasks; while introduction of MRS alleviates this problem.

Contribution 5. Presenting compositional analysis for MRS
In addition to previously mentioned sources of interference, tasks can also experience interference due to the timing constraints of shared memory.

In this contribution we address challenge C4, and we describe the problem of achieving composability of independently developed real-time subsystems to be executed on a multi-core platform. First, we evaluate existing work for achieving real-time predictability on multi-cores and illustrate their lacking with respect to composability. Second, we extend traditional compositional analysis for the multi-resource server. The multi-resource servers are useful to provide partitioning, composability and predictability in both hard and soft real-time systems. In this contribution we outline a theoretical framework to provide hard real-time guarantees for tasks executing inside a multi-resource server. We present a local schedulability analysis technique to assess the composability of subsystems containing hard real-time tasks. Using the compositional analysis technique, the system schedulability is checked by composing the subsystems interfaces which abstracts the resource demand of subsystems [7]. As previously described, tasks can also experience interference due to the timing constraints of the shared DRAM and the variable access time of DRAM. We present a worst case delay analysis of DRAM memory requests for COTS multicore architectures and incorporate this into the schedulability analysis for multi-resource servers. This contribution is presented in Papers E and F.

Note that a system consisting of a set of components (subsystems) is said to be composable w.r.t. some properties, if properties that have been established during the development of each component in isolation do not change when the components are integrated [34]. As mentioned before, we focus on the schedulability of tasks as the main timing property, thus a system is composable if the schedulability of tasks that have been validated during the development of each component is guaranteed when the components are integrated.

3.2 Challenges recapitulated

In this section we discuss: to what extent the research results and included papers meet the challenges presented in Section 2.3. We also comment on the validity of our results.

Challenge C1 has a broad scope and is realized through multiple implementations and papers. First, HSF support in FreeRTOS is described in Paper B and Paper C. Second, the realization of RVN is described in Paper C, and third, the MRS implementation is presented in Paper D.

3.2.1 HSF implementation

To address C1.1 challenge, we support FreeRTOS with hierarchical scheduling in Paper A. To develop an efficient HSF implementation with less overhead and to get better utilization of the system, a number of design considerations are made as explained in Section 6.4.6 and are addressed in Section 6.5.3. Experimental evaluation of the implementation for temporal isolation and faults containment (i.e., temporal errors are contained within the faulty component only and their effects are not propagated to the other components in the system) proves the hypothesis of predictable integration. Experiments are performed considering heavy-load and over-load situations for viability and efficiency. Moreover, overheads of hierarchical scheduling (i.e., tick handler, global scheduler, and task context-switch) are measured, see Section 6.6.2. The results reveal that the design decisions made lead to an efficient implementation.

Paper B addresses C1.2 challenge of migrating a legacy system in an HSF-setup. The presented approach is validated using a proof-of-concept case study that shows a successful integration of a legacy system (i.e., originally developed to execute as a stand-alone FreeRTOS application) along with the newly developed components (i.e., developed for HSF).

Validity: We test and validate the implementation by experimental results. Since other existing HSF implementations for unicore platforms use 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/compare different HSF implementations or different resource sharing protocols for HSF. For that reason we have implemented only two-level fixed-priority scheduling and one global resource locking protocol (HSRP). Another limitation is the execution of an example case study (instead of a larger industrial one). The case study is also limited to the FreeRTOS, since implementations of the proposed solutions are done for FreeRTOS operating system. Although the proposed solutions to integrate legacy system in HSF-setup are generic and can be implemented in any OS.

3.2.2 RVN and its realization

Paper C presents a proof-of-concept for the realization of our idea of RVN in the ProCom component model. It addresses C1.1, C1.2 and C2 by performing a case study and visualizing the execution traces. We test the system for fault containments, predictability in component's integration and reuse of components. In the experiments, the task set of each RVN is executed within a server using the specified resources 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. All these properties contribute to the predictability of RVNs. The increased predictability during component's integration further results in making the RVNs a reusable entity, as presented in results in Section 8.9.

Validity: We explain how to integrate RVN in only three component models [5]. We cannot claim that the idea of virtual node is applicable in general. We realize our idea in only ProCom component model. Another limitation is the execution of a simple case study (instead of a larger industrial one) due to the immaturity of the PRIDE tool.

3.2.3 MRS and its realization

Paper D addresses challenges C1 and C3 by implementing MRS which partitions CPU and memory-bus resources on multicore platforms. The experimental results reveal the MRS's suitability to execute applications in a predictable manner on a multi-core platform by limiting interference between applications running on different cores. The case study reveals that MRS is also suitable to migrate the legacy unicore code on multicore platform.
Validity: The results demonstrate that scheduling alone (i.e., controlling the allocation of resources over time) is not enough to achieve complete timing isolation. To achieve complete isolation on multicores, the space contention should also be controlled. Space contention arises due to the shared caches and memory [11]. The MRS should be adapted with space partitioning (i.e., partitioning of cache and memory) to control space contention. The underlying framework of MRS, ExSched framework [35, 36], includes some overheads by itself, in spite of the advantage of providing a kernel modification free solution. A kernel modification could have better performance.

3.2.4 Compositional analysis of the MRS

Challenge C4 is addressed in Papers E and F by presenting the analysis of the newly developed MRS. The analysis presented in Paper E safely bounds the memory contention for DDR DRAM memory controller that are commonly used in COTS multicore architectures and is used in the analysis of MRS. An experimental study is performed to investigate the correlation between the server budgets and the impact of different server periods on server-budgets.
Validity: The DRAM analysis is provided for COTS memory controllers considering only one rank and a single channel. Modern architectures are coming with multiple channels. We consider that the core stalls until the cache-line is fetched from memory. If multiple queues are present, then multiple commands can be handled, depending upon the availability of free space in the queue. We have explored the source of pessimism in our analysis which we intend to remove from the analysis in future.

Chapter 4

Conclusions and future work

This chapter concludes contributions of this dissertation and discusses future research directions.

4.1 Summary

In this dissertation, we have focused on preserving temporal properties of real-time software components during their integration and reuse. We have developed methods to integrate hierarchical scheduling within CBSE and have applied this technique to execute software components as servers within a two-level hierarchical scheduling framework (HSF). Moreover, we have adapted hierarchical scheduling for COTS multicore platforms on which the scheduling of real-time tasks is inherently unpredictable due to contention for shared physical resources.

We have implemented a two-level HSF in an open source real-time operating system, FreeRTOS, to support temporal isolation among real-time components. We have focused on being consistent with the underlying operating system by doing minimal changes and have kept the original FreeRTOS API semantics. We have enhanced the implementation by including a legacy server to execute a legacy code within a server. It includes adapting the original OS API with the HSF-API, and the sharing both hardware and software resources of the system among legacy and other components. We have tested our implementations and performed experimental evaluations on EVK1100 AVR based 32-bit micro-controller. Our results show a successful migration of the legacy system in an HSF-setup. The experimental validations of the implementations using heavy-load and over-load situations reveal temporal isolation among components. We have also measured overheads of executing hierarchical scheduler.

We have presented a novel executable component, *Runnable Virtual Node (RVN)* to address the challenges of preserving internal temporal requirements within components during their integration and reuse. The realization of RVN embeds the hierarchical scheduling technique within the software component using a two-level deployment process, thereby attaining temporal and functional isolation. Hence using runnable virtual nodes, a complex embedded system can be developed as a set of well-defined reusable components encapsulating both functional and timing properties. The proof-of-concept case studies executed on a micro-controller demonstrate the preserving of real-time properties within software components for predictable integration and reusability of legacy code, in both hierarchical scheduling and RVN contexts. Our work is based on the ProCom component-technology executing our HSF implementation on FreeRTOS. However, we believe that the concept is also applicable to commercial component technologies like AADL and AUTOSAR.

We have moved a significant step ahead towards expanding the resource reservation concept for COTS multicore platform on which the scheduling of real-time tasks is inherently unpredictable due to the contention for multiple shared physical resources [11]. It resulted in proposing and implementation of a novel type of server, called *Multi-Resource Server (MRS)* which controls the access to both CPU and memory-bus bandwidth resources in composing multiple real-time components on a multicore platform. The MRS provides temporal isolation both between tasks running on the same core, as well as, between tasks running on different cores. The results indicated that the MRS is suitable to migrate unicore legacy applications and to execute the legacy application with enough resources. Further, we have provided the compositional schedulability analysis for MRS to make it suitable for hard real-time systems. We have also performed experiments to study different properties of the newly developed MRS. It includes a correlation between the server budgets and the impact of different server periods on server-budgets.

4.2 Future research directions

In this section, we discuss future research directions that build on the research performed in this dissertation.

We see a big potential with RVN and the future development of it. Starting from general issues, we have realized our RVN concept in the ProCom component model using the PROGRESS Integrated Development Environment (PRIDE) tool by means of a proof-of-concept case study. The PRIDE tool is evolving and the automatic synthesis part is not fully mature. It could be interesting to do research on this part and then conduct a larger industrial case study on it. We believe that our concept is applicable also to commercial component technologies like AADL, AUTOSAR [5], and to realize the concept in these component technologies would be interesting and useful for the industry. Further, RVN concept can be extended for multicore platforms. It requires extensions at modeling realm by allocating the memory-bus bandwidth of each RVN, and executing the MRS implementation at executable realm. Another possible extension is to support virtual communication-busses using server-based scheduling techniques e.g., CAN [37] and Ethernet [38] in PRIDE tool. This will allow development, integration and reuse of distributed components using a set of virtual nodes and buses.

With respect to the MRS, we have only focused on *access contention* that results in temporal interference. Considering *space contention /space interference* issues in the implementation could be another interesting direction [11]. Space contention mainly arises at Last-Level Cache (LLC) and main memory (Dynamic RAM (DRAM)).

To achieve cache space isolation, one proposal could be a software-based implementation of *cache partitioning through physical page allocation* (also called *OS-based cache coloring*) [39, 40, 41, 42] and its integration with MRS running on the multicore platform. Cache-partitioning techniques divide the large shared physically-indexed cache into smaller private caches, and then allows allocating partitions of caches to components. Another method could be to bound caches by some static analysis technique [43, 44] and to incorporate the cache analysis into MRS analysis.

To achieve memory isolation, a software-based DRAM bank partitioning approach for MRS could be implemented, to control the memory contention and interference problems without any hardware modification to memory controllers. Some DRAM partitioning approaches to partition the banks among tasks (or cores) has been implemented [45, 41, 24].

We have explored the source of pessimism in the analysis of MRS and, in future, research could be done to remove some pessimism from the analysis. Another future direction is to find an algorithm to calculate the optimum budgets for both resources of the MRS and to find a smart online algorithms to assign the unused capacity of one resource to another server to improve the overall average response times. Another interesting direction could be to make the hierarchical scheduling adaptive in nature by implementing mode switches into the hierarchical scheduling. We have started from adapting the CPU time [46, 47] for unicore platforms, and next step would be to adapt it for multicore platform by including memory issues.

Further possibilities to be investigated in future are specific for each paper and are presented in respective papers.

Chapter 5

Overview of dissertation

This chapter summarize the included papers, my personal contribution to these, and the most important assumptions and limitations of our results.

5.1 Summary of papers

A short description and contribution of the papers and reports included in this dissertation is given here. For each paper there is also an account of my personal contribution.

Technical reports included in this dissertation are all based on extension of peer-reviewed papers. For these contributions, we describe these extension and the relation to the original published papers.

Paper A. "Support for hierarchical scheduling in FreeRTOS"

Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, Syed Mohammed Hussein Ashjaei, Sara Afshar. In Proceedings of the 16^{th} IEEE International Conference on Emerging Technologies and Factory Automation (ETFA' 11). 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.

Personal contribution: I am 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 two students who were responsible for the implementation part.

Paper B. "Support for legacy real-time applications in an HSF-enabled FreeRTOS"

Rafia Inam, Moris Behnam, Mikael Sjödin. MRTC report ISSN 1404-3041 ISRN MDH-MRTC-295/2014-1-SE, Mälardalen University, November, 2014. Submitted to the Journal of systems Architecture (JSA), 2014.

Short Summary: This paper presents runtime support to consolidate legacy and new real-time applications, running a single instance of a real-time operating system (RTOS), and sharing system resources. In this context, we leverage a hierarchical scheduling framework (HSF) to provide temporal partitions for different applications, supporting their independent development and real-time analysis. These temporal partitions paves way for predictable integration. In particular, the paper focuses on a constructive element, we call the legacy server, that allows executing code that is unaware of the temporal partition within which it is deployed. Furthermore, we discuss the challenges that need to be addressed to execute a legacy application in an HSF without modifications to the original code. We focus on the challenge of enabling sharing system resources, both hardware and software, as typically found in most embedded software-systems. We propose a novel solution based on wrappers for RTOS system calls.

We implement our ideas in a concrete implementation on FreeRTOS, taking advantage of a prior HSF implementation. The validation is performed by a proof-of-concept case study that shows successful integration of a legacy application that uses shared resources in a system that executes other applications.

Personal contribution: I was the initiator and the main author of this paper. All co-authors have contributed with valuable discussions and reviews.

Extended version: The paper included in this dissertation extends our previous work that has been published as a full paper in the 7^{th} International Workshop on Operating Systems Platforms for Embedded Real-Time Applications, which provided resource sharing protocols' implementation for HSF [48]. To be precise, the work in paper B completes the legacy system's consolidation with other real-time components in an HSF environment by developing new wrappers for the original API of operating system and performing detailed experimental validations.

Paper C. "Predictable integration and reuse of executable real-time components"

Rafia Inam, Jan Carlson, Mikael Sjödin, Jiří Kunčar. In the Journal of Systems and Software (JSS), Vol 91, pages 147-162, May, 2014.

Short Summary: We present the concept of runnable virtual node (RVN) as a means to achieve predictable integration and reuse of executable real-time components in embedded systems. A runnable virtual node is a coarse-grained software 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 behaviour is preserved when integrated with other components or when reused in a new environment. Our realization of RVN 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, i.e. deploying functional entities to RVNs and then deploying RVNs to physical nodes, and thus also gives development benefits with respect to composability, system integration, testing, and validation. In addition, we have implemented a server-based inter-RVN communication strategy to not only support the predictable integration and reuse properties of RVNs by keeping the communication code in a separate server, but also increasing the maintainability and flexibility to change the communication code without affecting the timing properties of RVNs. We have applied our approach to a case study, implemented in the ProCom component technology executing on top of a FreeRTOS-based hierarchical scheduling framework and present the results as a proof-of-concept.

Personal contribution: I am the initiator and principal author to all parts in this paper. Jiří was responsible of integrating HSF code into PRIDE tool. I have contributed in the design and execution of the case study on the target platform using AVR Studio and performed all the tests and experiments. Jiří developed the EELAP tool to compute the end-to-end latencies of both communication strategies [33]. All coauthors have contributed with valuable discussions and reviews.

Paper D. "The Multi-Resource Server for predictable execution on multicore platforms"

Rafia Inam, Nesredin Mahmud, Moris Behnam, Thomas Nolte, Mikael Sjödin. 20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'14), pages 1-10, April, 2014.

Short Summary: In this paper we present an implementation and demonstration of the Multi-Resource Server (MRS) which enables predictable execution of real-time applications on multi-core platforms. The MRS provides temporal isolation both between tasks running on the same core, as well as, between tasks running on different cores. The latter could, without MRS, interfere with each other due to contention on a shared memory bus. We demonstrate that MRS can be used to encapsulate legacy systems and to give them enough resources to fulfill their purpose. In our case study a legacy media-player is integrated with several resource-hungry tasks running at a different core. We show that without MRS the media-player starts to drop frames due to the interference from other tasks; while introduction of MRS alleviates this problem. Another part of our demonstration shows how traditional periodic real-time tasks can be kept schedulable even when tasks with high memory-demand are added to the system.

Personal contribution: I am the main driver and author to all parts in this paper. I have contributed in the design of MRS implementation on ExSched platform, design of the case study and synthetic experiments. Nesredin was responsible for implementing multicore functionality of the MRS and for executing it on the target platform. All other coauthors have contributed with valuable discussions and reviews.

Paper E. “Worst case delay analysis of a DRAM memory request for COTS multicore architectures”

Rafia Inam, Moris Behnam, Mikael Sjödin. Seventh Swedish Workshop on Multicore Computing (MCC’14), November, 2014.

Short Summary: Dynamic RAM (DRAM) is a source of memory contention and interference problems on commercial of the shelf (COTS) multicore architectures. Due to its variable access time, it can greatly influence the task’s WCET and can lead to unpredictability. In this paper, we provide a worst case delay analysis for a DRAM memory request to safely bound memory contention on multicore architectures. We derive a worst-case service time for a single memory request and then combine it with the per-request memory interference that can be generated by the tasks executing on same or different cores in order to generate the delay bound.

Personal contribution: I am the initiator and author to all parts in this paper. All other coauthors have contributed with valuable discussions and reviews.

Paper F. “Compositional analysis for the Multi-Resource Server – a technical report”

Rafia Inam, Moris Behnam, Thomas Nolte, Mikael Sjödin. MRTC report ISSN 1404-3041 ISRN MDH-MRTC-283/2014-1-SE, Mälardalen University, September, 2014. Submitted for conference publication.

Short Summary: The Multi-Resource Server (MRS) technique has been proposed to enable predictable execution of memory intensive real-time applications on COTS multi-core platforms. It uses resource reservation approaches in the context of CPU-bandwidth and memory-bus bandwidth reservations to bound the interferences between the applications running on the same core as well as between the applications running on different cores. In this paper we present a complete compositional schedulability analysis for the Multi-Resource Server technique. Based on the proposed analysis, we further provide an experimental study that investigates the behaviour of the MRS and identifies the factors that contribute mostly on the overall system performance.

Personal contribution: I am the main driver and author to this paper. Moris has contributed with the local schedulability analysis section of the preliminary work [49]. All the coauthors have contributed with valuable discussions and reviews.

Extended version: This paper extends our previous work that is published the journal ACM SIGBED Review, special issue on 5th Workshop on Compositional Theory and Technology for Real-Time Embedded Systems, Vol 10, nr 3, Oct, 2013. [49]. We update the local analysis and present a complete and composable global schedulability analysis for both resources (CPU- and memory-bus bandwidth) of the MRS. Further, we provide a study that investigates the behavior of the MRS and brings insight on how these both resources relate to each other. The preliminary work [49] described the initial local schedulability analysis of MRS and did not address the global schedulability analysis and lacked an investigation study.

5.2 Delimitations of the research

The research presented in this dissertation has been done under some assumptions and restrictions. This section reiterates some of the most important assumptions and limitations of our research.

Real-time systems:
Most articles included in this dissertation assume hard real-time systems. The paper D is an exception which is developed using a Linux-based common framework ExSched [35]. We relax the assumption to the level of soft real-time systems in this particular work.

Components model:
Although we have focused on ProCom, the general ideas regarding component based software engineering is applicable to other component models (such as AADL and AUTOSAR)

Components:
We have worked on software components at the executable-level for the unicore platform only.

Multicore architecture:
For papers D, we execute results on a multicore architecture, an Intel core duo processor. The hardware is not commonly used in hard real-time systems and has unpredictable behavior coming from hardware functionality in caches, frequency scaling, prefetching instructions etc. For our

results, we have disabled the options of frequency scaling, prefetching instructions, and we have cleared the cache before executing the results. Despite this, the caches and memory are still shared, therefore, we assume the level of soft real-time systems in this particular work.

Scheduling algorithm:
We assume fixed-priority preemptive scheduling at both levels of hierarchies in HSF.

Partitioning:
Our work includes time partitioning only, i.e., CPU and memory-bus partitioning. On multicore architectures, the space partitioning (i.e., cache and memory partitioning) is also needed to provide complete isolation.

HSF implementation on FreeRTOS:
In this work, we have not focused to evaluate/compare different HSF implementations.

Resource sharing:
We have implemented one global resource locking protocol i.e., HSRP on unicore platform. Comparing different resource locking protocols is not the main focus of this work. Resource sharing on multicore platforms is not considered.

Case studies:
Another limitation is the execution of case studies only for the proof-of-concept purpose (instead of a larger industrial one).

Bibliography

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

[2] R. N. Charette. "This car runs on code". *Spectrum*, 46(2), 2009. "http://spectrum.ieee.org/greentech/advanced-cars/this-car-runs-on-code".

[3] Charlotte Adams. Reusable software components: Will they save time and money?, 2005. [Online]. Available: http://www.aviationtoday.com/, last checked: 20.03.2013.

[4] Charlotte Adams. Product focus: Cots operating systems: Boarding the boeing 787, 2005. [Online]. Available: http://www.aviationtoday.com/, last checked: 20.03.2013.

[5] R. Inam, J. Mäki-Turja, J. Carlson, and M. Sjödin. Virtual Node – To Achieve Temporal Isolation and Predictable Integration of Real-Time Components. *International Journal on Computing (JoC)*, 1(4), January 2012.

[6] Z. Deng and J. W.-S. Liu. Scheduling real-time applications in an open environment. In *Proc. 18^{th} IEEE Real-Time Systems Symposium (RTSS' 97)*, pages 308–319, December 1997.

[7] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proc. 24^{th} IEEE Real-Time Systems Symposium (RTSS' 03)*, pages 2–13, December 2003.

[8] T. Nolte, I. Shin, M. Behnam, and M. Sjödin. A Synchronization Protocol for Temporal Isolation of Software Components in Vehicular Systems.

IEEE Transactions on Industrial Informatics, 5(4):375–387, November 2009.

[9] 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.

[10] L. Abeni, L. Palopoli, C. Scordino, and G. Lipari. Resource reservations for general purpose applications. *IEEE Transactions on Industrial Informatics*, 5(1):12–21, Feb 2009.

[11] R. Inam and M. Sjödin. Combating unpredictability in multicores through the Multi-Resource Server. In *Workshop on Virtualization for Real-Time Embedded Systems (VtRES'14)*. IEEE, September 2014.

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

[13] 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, December 2006.

[14] 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)*, pages 74–82, September 2010.

[15] 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.

[16] 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 (CBSE' 08)*, pages 310–317, October 2008.

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

[18] 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' 11)*, pages 129–138. ACM, June 2011.

[19] 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 (ECRTS' 10) WiP Session*, pages 17–20, July 2010.

[20] Rafia Inam. *Towards a Predictable Component-Based Run-Time System*. Number 145 in Mlardalen University Press Licentiate Theses. Licentiate thesis, Västerås, Sweden, January 2012.

[21] T. 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)*, pages 57–60, August 2011.

[22] J. P. Lehoczky, L. Sha, and J.K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proc. 8^{th} IEEE Real-Time Systems Symposium (RTSS' 87)*, pages 261–270, December 1987.

[23] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems*, 1(1):27–60, June 1989.

[24] H. Yun, R. Mancuso, Z-P. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *Proc. 20^{th} IEEE Real-Time Technology and Applications Symposium (RTAS' 14)*, April 2014.

[25] Z-P. Wu, Y. Krish, and R. Pellizzoni. Worst case analysis of DRAM latency in multi-requestor systems. In *Proc. 34^{th} IEEE Real-Time Systems Symposium (RTSS' 13)*, pages 372–383, December 2013.

[26] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. R. Rajkumar. Bounding memory interference delay in COTS-based multicore systems. In *Proc. 20^{th} IEEE Real-Time Technology and Applications Symposium (RTAS' 14)*, pages 145–154, April 2014.

[27] M. Shaw. The coming-of-age of software architecture research. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE' 01)*, pages 656–664, 2001.

[28] Richard Barry. *Using the FreeRTOS Real Time Kernel*. Real Time Engineers Ltd., 2010.

[29] L. Sha, J.P. Lehoczky, and R. Rajkumar. Solutions for some Practical problems in Prioritised Preemptive Scheduling. In *Proc. 7th IEEE Real-Time Systems Symposium (RTSS' 86)*, pages 181–191, December 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] R. Inam, J. Mäki-Turja, M. Sjödin, S. M. H. Ashjaei, and S. Afshar. Support for Hierarchical Scheduling in FreeRTOS. In *16th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA' 11)*, pages 1–10, 2011.

[32] Jiří Kunčar. End-to-End Latency Analyzer for ProCom - EELAP, March 2013. https://github.com/jirikuncar/eelap/.

[33] Jiří Kunčar, R. Inam, and M. Sjödin. End-to-End Latency Analyzer for ProCom - EELAP. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-272/2013-1-SE, Mälardalen University, School of Innovation, Design and Engineering, 2013.

[34] H. Kopetz and N. Suri. Compositional design of RT systems: A conceptual basis for specification of linking interfaces. In *6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC' 03)*, pages 51–57, May 2003.

[35] M. Åsberg, T. Nolte, S. Kato, and R. Rajkumar. ExSched: an external cpu scheduler framework for real-time systems. In *18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA' 12)*, pages 240–249, August 2012.

[36] R. Inam, J. Slatman, M. Behnam, M. Sjödin, and T. Nolte. Towards implementing Multi-Resource Server on multi-core Linux platform. In *18th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA' 13), WiP*, pages 1–4, September 2013.

[37] T. Nolte, M. Nolin, and H. Hansson. Real-Time Server-Based Communication for CAN. *IEEE Transaction on Industrial Electronics*, 1(3):192–201, April 2005.

[38] 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.

[39] J. Liedtke, H. Härtig, and M. Hohmuth. OS-controlled cache predictability for real-time systems. In *Proc. 3rd IEEE Real-Time Technology and Applications Symposium (RTAS' 97)*, pages 213–224, June 1997.

[40] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *IEEE 14th International Symposium on High Performance Computer Architecture, (HPCA 08)*, pages 367–378, February 2008.

[41] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *Proc. 19th IEEE Real-Time Technology and Applications Symposium (RTAS' 13)*, pages 45–54, April 2013.

[42] A. Wolfe. Software-based cache partitioning for real-time applications. *J. of Comp. Sofw. Engi.*, 2(3):315–327, Mar 1994.

[43] S. Schliecker and R. Ernst. Real-time Performance Analysis of Multiprocessor Systems with Shared Memory. *ACM Transactions in Embedded Computing Systems*, 10(2):22:1–22:27, January 2011.

[44] D. Dasari and B. Anderssom and V. Nelis and S.M. Petters and A. Easwaran and L. Jinkyu . Response Time Analysis of COTS-Based Multicores Considering the Contention on the Shared Memory Bus. In *Proc. of the IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom '11)*, pages 1068 – 1075, November 2011.

[45] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *Proc. of the 21st International Conf. on Parallel Architectures and Compilation Techniques (PACT' 12)*, pages 367–376, 2012.

[46] R. Inam, M. Sjödin, and R. J. Bril. Implementing Hierarchical Scheduling to support Multi-Mode System. In *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12), WiP*, pages 319 – 322, June 2012.

[47] Rafia Inam, Mikael Sjödin, and Reinder J. Bril. Mode-change mechanisms support for hierarchical freertos implementation. In *18th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA' 13)*, pages 1 – 10, September 2013.

[48] R. Inam, J. Mäki-Turja, M. Sjödin, and M. Behnam. Hard Real-time Support for Hierarchical Scheduling in FreeRTOS. In *7th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT' 11)*, pages 51–60, July 2011.

[49] M. Behnam, R. Inam, T. Nolte, and M. Sjödin. Multi-core composability in the face of memory bus contention. *ACM SIGBED Review, Special Issue on 5th Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, 10(3), 2013.

II

Included papers

Chapter 6

Paper A: 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.

keywords: Hierarchical scheduling, Real-time open systems, Hierarchical scheduling framework, Fixed-priority scheduling.

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.

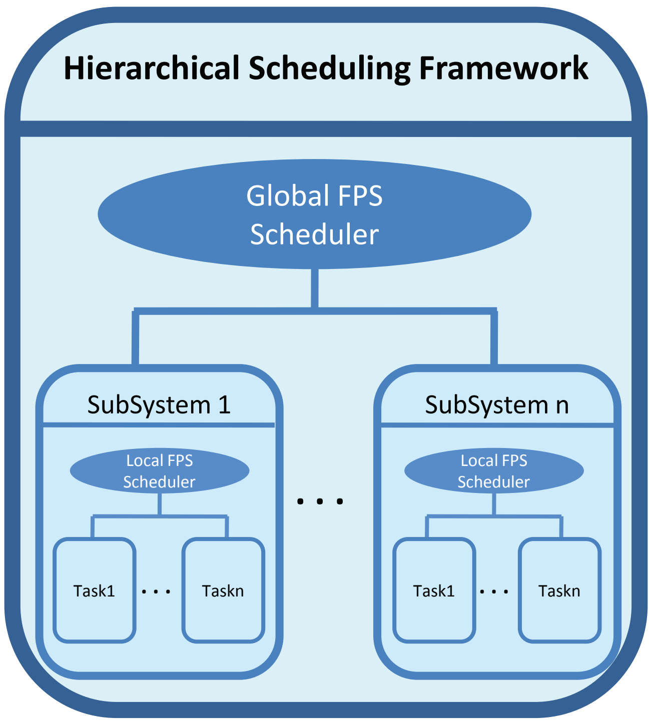


Figure 6.1: Two-level Hierarchical Scheduling Framework

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

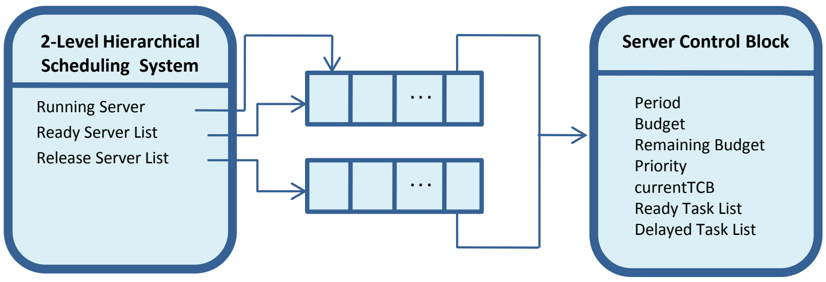


Figure 6.2: Data structures for active and inactive servers

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.

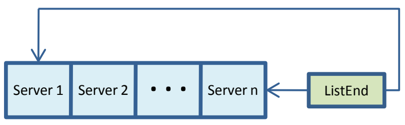


Figure 6.3: The structure of ready-server and release-server lists

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.

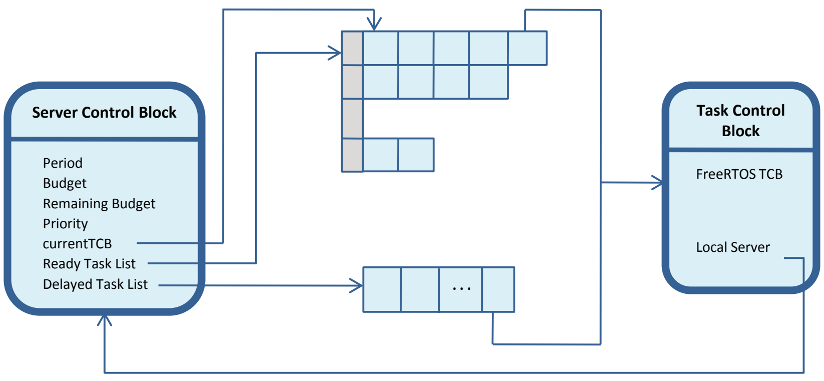


Figure 6.4: Data structures for ready and delayed tasks

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.

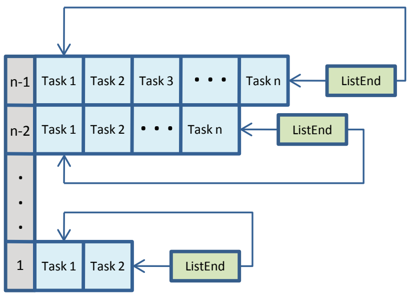


Figure 6.5: The structure of ready-task list

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 `prvChooseNext DeferrableServer()` 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

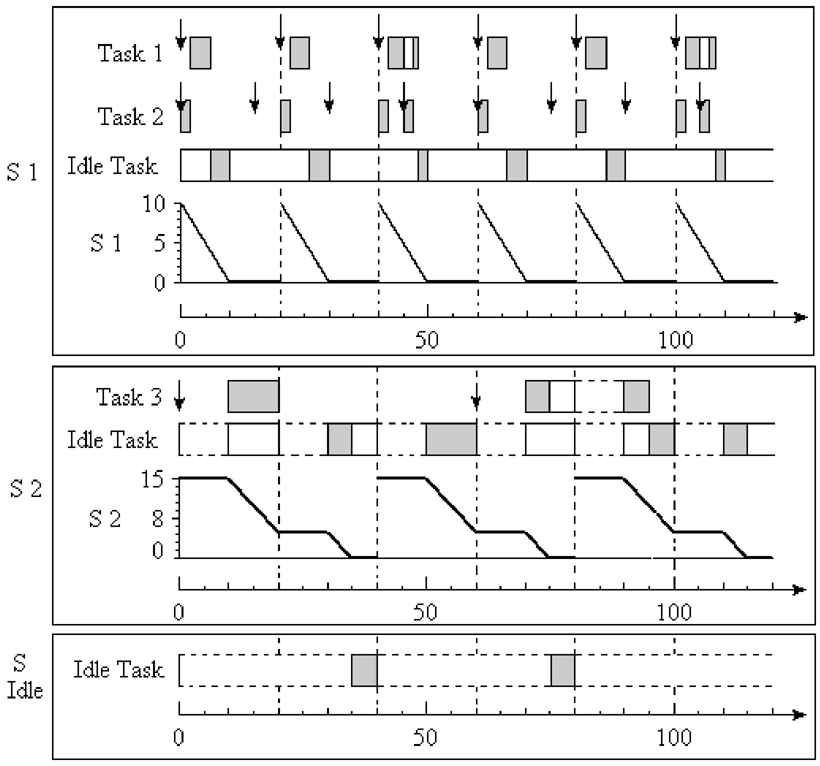


Figure 6.6: Trace for idling periodic servers

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

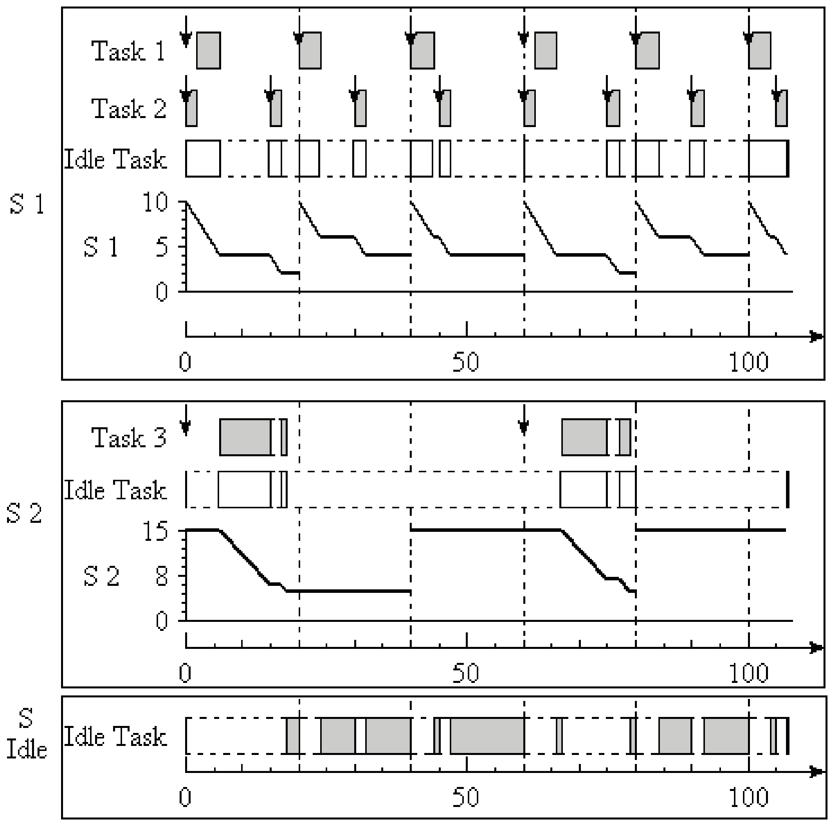


Figure 6.7: Trace for deferrable servers

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.

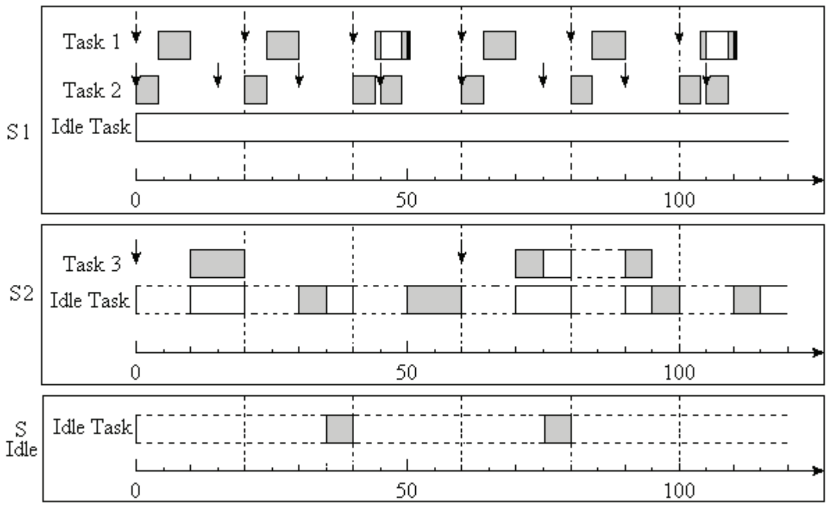


Figure 6.8: Trace showing temporal isolation among idling servers

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, *pxCreated-Task, 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' 97)*, pages 308–319, December 1997.

[4] FreeRTOS web-site. [Online]. Available: 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 (ECRTS' 10) 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] D. Kim, Y-H 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' 03)*, pages 2–13, December 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' 99)*, December 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' 00)*, 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, September 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, December 2002.

[13] R. I. Davis and A. Burns. Hierarchical Fixed Priority Pre-emptive Scheduling. In *Proc. 26th IEEE Real-Time Systems Symposium (RTSS' 05)*, pages 389–398, December 2005.

[14] S. Saewong and R. Rajkumar. Hierarchical reservation support in resource kernels. In *Proc. 22th IEEE Real-Time Systems Symposium (RTSS' 01)*, December 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] M. Behnam, T. Nolte, I. Shin, M. Åsberg, and R. 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. 7th IEEE Real-Time Systems Symposium (RTSS' 86)*, pages 181–191, December 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] M. Behnam, T. Nolte, M. Sjödin, and I. 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 B: Support for legacy real-time applications in an HSF-enabled FreeRTOS

Rafia Inam, Moris Behnam, Mikael Sjödin
MRTC report ISSN 1404-3041, ISRN MDH-MRTC-295/2014-1-SE, Mälardalen Real-Time Research Center, Mälardalen University, November, 2014.
(submitted for journal publication).

Abstract

This paper presents runtime support to consolidate legacy and new real-time applications, running a single instance of a real-time operating system (RTOS), and sharing system resources. In this context, we leverage a hierarchical scheduling framework (HSF) to provide temporal partitions for different applications, supporting their independent development and real-time analysis. These temporal partitions paves way for predictable integration. In particular, the paper focuses on a constructive element, we call the legacy server, that allows executing code that is unaware of the temporal partition within which it is deployed. Furthermore, we discuss the challenges that need to be addressed to execute a legacy application in an HSF without modifications to the original code. We focus on the challenge of enabling sharing system resources, both hardware and software, as typically found in most embedded software-systems. We propose a novel solution based on wrappers for RTOS system calls.

We implement our ideas in a concrete implementation on FreeRTOS, taking advantage of a prior HSF implementation. The validation is performed by a proof-of-concept case study that shows successful integration of a legacy application that uses shared resources in a system that executes other applications.

keywords: real-time systems, hierarchical scheduling, legacy application reuse, applications integration.

7.1 Introduction

The trend of software reuse is observed in many industrial embedded software applications. The reuse of legacy applications is an answer to industrial challenges like development cost, time to market, and increasing complexity. For instance, the new Boeing 787 ”Dreamliner” is a recent example with a significant proportion of reused modules from another Boeing airplane [1, 2]. Furthermore, many industrial systems are developed in an evolutionary fashion, reusing applications from previous versions or from related products. It means that applications are reused and re-integrated in new environments.

Integration of real-time applications, defined as consistent sets of concurrent time-constrained tasks, can be explained as the mechanism of wiring applications together [3]. For real-time embedded systems, integrating legacy and other real-time applications must achieve both (1) functional correctness and (2) satisfy extra-functional properties, particularly timing properties. The temporal behaviour of real-time software applications poses difficulties during integration. Upon integration, tasks of one application affect the scheduling of tasks of other applications. This means that for an embedded system with real-time constraints; an application that is found correct during unit verification may fail due to a change in temporal behaviour when integrated in a system.

Virtualization is a resource-management technique to mitigate these problems by partitioning the system resources, such as processor, memory or network, in a way that provides the illusion of a full resource but with a fractional capacity [4]. Using virtualization, a CPU resource is partitioned in several smaller virtual machines (VMs), each running a separate operating system instance either without any modification, e.g. KVM-based solutions [5], or with modifications, e.g. Xen [6]. However, executing multiple operating systems maybe undesired and the performance overhead introduced by virtualization is a big challenge for resource constrained embedded hardware nodes particularly smaller microcontrollers. Furthermore, system administration of virtualization can become a time-consuming task due to complex configuration interactions between supposedly disjoint applications. A comparatively lightweight technique to allow the partitioning of an OS environment into multiple temporal partitions and to execute a separate real-time application in each partition is *OS virtualization*. In this approach, only a single instance of an OS executes in the system, supporting all applications, being better suited for the resource constrained embedded hardware. OS virtualization is typically implemented using *server-based scheduling*, which was generalized into the *Hierarchical Scheduling Framework (HSF)* [7, 8].

HSF offers an efficient mechanism (i) to provide predictable integration of applications by rendering temporal partitioning among them [9], (ii) to support independent development and analysis of real-time applications [8], and (iii) to provide analysis of integrated applications at the system level [8, 10]. These advantages of HSF could provide an even better leverage for reusing real-time legacy applications. However, most of the existing research focus is on providing analysis tools and algorithms in order to enable predictable reusability of applications [8, 9]. Similar approaches have been proposed targeting specifically legacy applications, even if the timing characteristics of the applications are not known in advance [11]. However, predictable reuse of legacy applications with HSF requires additional runtime support which, to the best of our knowledge, has not been investigated previously.

In this paper we provide an implementation support to execute legacy applications in servers within a two-level HSF. Our method is based on creating a legacy server for each legacy application. Then, for each application, we allocate its tasks to the respective server. Thus a legacy server encapsulates a legacy application and becomes a container for a set of legacy tasks. The use of legacy servers upholds the independent development of legacy applications from the rest of the system, encapsulates internal temporal properties of the legacy applications, and ensures the predictable temporal behaviour of the system. To support resource sharing among tasks of the same server (called *local resource sharing*) and among tasks of different servers (called *global resource sharing*), we implement two resource sharing protocols: *Stack Resource Policy (SRP)* [12], and *Hierarchical Stack Resource Policy (HSRP)* [13, 14] respectively.

As target system we chose the FreeRTOS operating system for which we already have a two-levels HSF implementation for independent applications [15]. In this paper we extend existing research with the following contributions:

- Identification of the challenges involved in running legacy applications (with no or minimal changes) in the hierarchical framework.
- A runtime support for reusing legacy real-time applications. This entails: (1) *an implementation of a legacy server*. (2) *the development of new wrappers for the original OS API* to support software and hardware resource sharing among legacy and other applications. (3) *implementations of resource sharing protocols*. We presented a preliminary implementation in [16] which is subsumed by this work.
- Experimental validation of the proposed solution and its implementa-

tion. We *apply a case study to evaluate the implementation* in terms of correctness and runtime overhead. A legacy application that uses FreeRTOS resource sharing API is executed within a legacy server to check (1) the automatic creation of legacy tasks and their execution within the legacy server, and (2) the correctness of wrappers. We also perform tests for HSRP implementation and wrappers for a hardware resource, that is shared between an external task and a legacy task. And finally, we *test and measure the performance* of our implementations for synchronization protocols at both levels on an AVR-based 32-bit board EVK1100 [17]. We also compare overheads of the wrapper against the original FreeRTOS API.

To the best of our knowledge, this is the first work to identify the challenges involved in executing a legacy application within an HSF and to provide an implementation of the needed execution support.
Organization of the paper: Section 7.2 describes the challenges in executing legacy applications in an HSF. Section 7.3 provides our system model. Section 7.4 gives a background on FreeRTOS and reviews the HSF implementation in FreeRTOS. Section 7.5 overviews the resource sharing techniques for HSF. Sections 7.6 and 7.7 present our implementation of resource sharing and legacy support respectively. Section 7.8 presents a case study of a legacy application, and Section 7.9 experimentally evaluates the behavior of a legacy application that uses resource sharing and presents overhead measures. Section 7.10 describes related work. Section 7.11 concludes the paper and finally, an Appendix lists the API and macros of the implementation.

7.2 Challenges in executing legacy applications in an HSF

Integrating a legacy application, originally developed for full CPU access, in a two-level hierarchical framework raises many challenges. Our goal is to make minimal changes in the legacy application, i.e., we exclude the possibility of changing the application code to convert the original calls to the OS API in calls to the HSF API. In fact, the legacy application is already tried-and-tested, and has been already deployed and executed, thus is more reliable to leave it as is. Moreover, making such changes in the legacy code is tedious, time consuming, and error prone.

The first challenge is to create a legacy server itself and execute legacy tasks inside. A new API is required to create the legacy server, create legacy tasks and assign legacy tasks to the server.

The second challenge is to execute the legacy application without modifying it. The code of the legacy application still calls the original OS API. However, executing the legacy application in the hierarchical environment requires the HSF API to be called instead. For example, in FreeRTOS the `xTaskCreate` system call is used for task creation, but the `xServerTaskCreate` system call is used in HSF for tasks creation within a server.

A third challenge arises when the legacy application accesses a shared resource and uses synchronization primitives of the OS. When the same code is executed in a legacy-server within a two-level HSF along with other servers, the resource which is shared among tasks of the same server, i.e. legacy-server, is considered as a local resource. It is important to create and retain the resource within the server, and tasks of other servers should not be allowed to access this resource. This is a requirement for the temporal isolation among servers in HSF. In addition, for the legacy application, resources that might be shared with other applications are considered as global resources and the HSRP-based resource access API should be used in this case. This adaptation could also be done by changing the respective system calls embedded in the legacy code to a convenient resource sharing protocol. However, changing the original code is error-prone and time consuming. Moreover, changing the synchronization protocol would change the semantics of the legacy application (e.g. changing semantics of PIP to SRP) which is undesired.

In addition, the choice of synchronization protocol to be used in HSF depends on whether the legacy application is sharing the resource with other applications or not. If it is not, the original system calls should be used i.e., the legacy application is granted exclusive access the processor resource. Otherwise, global synchronization primitives should be used instead. Again, to avoid changing the legacy code, we keep its unawareness of whether a resource is local or global and we delay this choice for deployment phase, relying on the operating system knowledge.

To overcome these challenges and execute the legacy application in HSF without modification, we develop wrappers around the original OS API. A wrapper is a middleware that separates the original OS API from the actual system calls code. Wrappers exhibit the same interface as of the original API, but extend these with some extra functionality to call the new system calls [18]. This allows invoking the new system calls from within the legacy application without changing the legacy code. The advantage of wrapping over conventional redevelopment is that it requires less effort and lower development cost while keeping the original code unchanged. Moreover, it retains the semantics of the original operating system code.

7.3 System model

In this paper, we consider a two-level HSF using the periodic resource model [8] in which a system $\mathcal{S}$ consists of a set of independently developed and analyzed subsystems S_s, each representing one application. The HSF can be viewed as a tree with one parent node and many leaf nodes as illustrated in Figure 7.1. The parent node is a *global scheduler* and leaf nodes are subsystems. Each subsystem S_s consists of its own internal set of tasks that are scheduled by a *local scheduler*, and is executed by a server. The global scheduler schedules the system and is responsible for dispatching the servers according to their resource reservations. The local scheduler of each subsystem then schedules its task set according to a server-internal scheduling policy. The system contains a set of *global shared resources*, shared among tasks of different subsystems, and each subsystem has a set of *local shared resources*, shared among tasks of the same subsystem. In the rest of this paper, we use the term subsystem and server interchangeably.

7.3.1 Subsystem model

Each subsystem S_s is specified by a timing interface $S_s = \langle P_s, Q_s, p_s, X_s \rangle$, where P_s is the period for that server ($P_s > 0$), Q_s is the capacity allocated periodically to the server ($0 < Q_s \leq P_s$), and X_s is the maximum execution-time that any subsystem-internal task may lock a global shared resource $0 < X_s \leq Q_s$. Each server S_s has a priority p_s. The idle server has lowest priority i.e. 0 in the system. At each instant during run-time, B_s represents a remaining budget, $B_s \leq Q_s$. During execution of a subsystem, B_s is decremented by one at every time unit until it depletes. When $B_s = 0$, the budget is depleted and S_s will be suspended until its next period when B_s is replenished with Q_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.

7.3.2 Task model

We consider a simple periodic task model represented by a set $\mathcal{T}$ of n number of tasks. Each task τ_i is represented as $\tau_i = \langle T_i, C_i, \rho_i, cs \rangle$, where T_i denotes

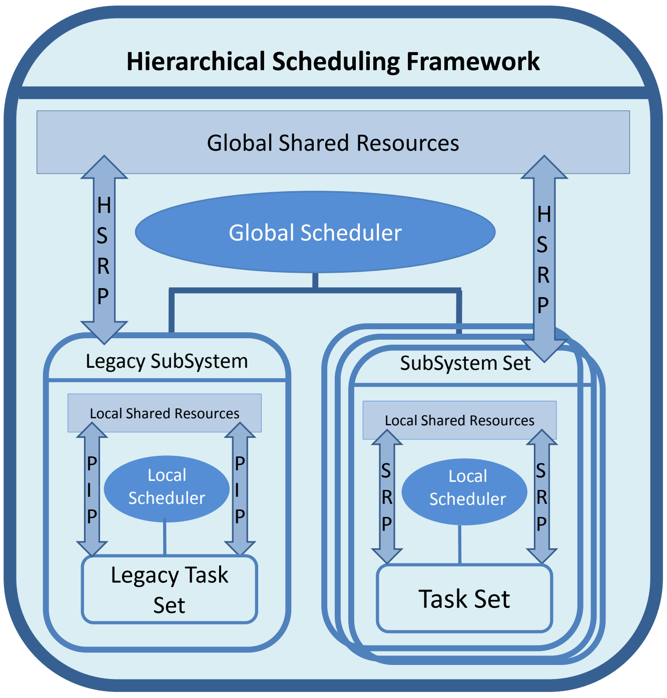


Figure 7.1: Two-level Hierarchical Scheduling Framework

the period of task τ_i with worst-case execution time C_i where $0 < C_i \leq T_i$, ρ_i as its priority, and cs is the set of critical section execution times of all resources that the task accesses. For simplicity, we do not consider the case of nested resource access in this paper. 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. For simplicity, the deadline for each task is equal to T_i.

7.3.3 Scheduling policy

We use fixed-priority preemptive scheduling (FPPS) at both global and local levels of scheduling. FPPS is flexible and is the de-facto industrial standard for task scheduling [19]. Our implementation supports shared priorities, which are then handled in FIFO order (both in global and local scheduling).

7.3.4 The legacy model

Each legacy application consists of a legacy task set, and a set of resources shared among those tasks. We assume that the legacy application is developed for FreeRTOS operating system and the source code is available. In HSF, the legacy application is now executed as a legacy server.

7.3.5 Summary of analytical framework

To perform real-time analysis techniques, a priori knowledge of tasks parameters and server parameters is required, which are generally not known for legacy applications. Given parameters of FreeRTOS tasks, a task's period and priority are derived for the legacy application. Given interfaces of tasks, the server interface can be derived using available technique [11] that not only identifies the execution requirements of unknown applications, but can also be used to self-tune the scheduling parameters of legacy applications by using feedback scheduling. However, the resource sharing among tasks is not considered in that approach.

Given the system model, analytical frameworks exist to perform the schedulability analysis that can be used to integrate the newly developed applications for HSF and the legacy application together [8, 14, 11]. Since analysis framework has been established, we complement it with practical implementation to allow it for practise. Note that the focus of this work is on integration and implementation; we leave the identification of blocking times for locked resources of legacy application as a future work.

7.4 FreeRTOS and its HSF implementation

This section presents the background on FreeRTOS and its synchronization primitives. Further it presents a brief overview of a HSF implementation in FreeRTOS. The HSF implementation is already presented in [15] and is included here for the sake of completeness.

7.4.1 FreeRTOS and its synchronization primitives

FreeRTOS is a portable, open source (licensed under a modified GPL), mini real-time operating system developed by Real Time Engineers Ltd [20]. It is ported to more than 20 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, resulting in a binary image between 4 to 9KB. Thus it is suitable for resource constraint micro-controllers. FreeRTOS kernel supports preemptive, cooperative, and hybrid scheduling. Using FPPS, tasks with the same priority are scheduled using the round-robin policy. It supports an arbitrary number of tasks, with both static and dynamic (changed at run-time) priorities, and 256 different priorities for tasks. Its scheduler runs at the rate of one tick per milli-second by default. It implements a very efficient task context-switch (i.e 10μs for the rate 1 milli-second).

FreeRTOS supports basic synchronization primitives like *binary, counting* and *recursive semaphore*, and *mutexes*. The mutexes employ priority inheritance protocol (PIP) [21], in which a lower priority task that is locking a shared resource inherits the priorities of all tasks that have higher priority and try to access the same resource. 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.

FreeRTOS implements all the above mentioned synchronization primitives using the message queues without buffering. The message queue structure `xQueue` is initiated at the creation of a semaphore or mutex and message queue API is called to handle synchronization among tasks. Each semaphore creates a separate queue to handle synchronization. `xSemaphorehandle` pointer points to a queue structure `xQueueHandle` created for that semaphore.

7.4.2 HSF implementation in FreeRTOS

A two-level HSF implementation on FreeRTOS [15] supports idling periodic [22] and deferrable servers [23]. *Idling periodic* means that tasks in the server execute and use the server's capacity until it is depleted. If the server has 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. *Deferrable server* means that tasks execute and use the servers capacity. 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 servers period, it will be served and consumes servers capacity until the capacity depletes or the servers 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.

To follow the periodic resource model [8], our servers and tasks are activated periodically. Servers replenish their budget Q_s every constant period P_s. Since FPPS is used at both global and local scheduling levels, a higher priority server/task can preempt the execution of lower priority servers/tasks respectively. A brief overview of the implementation [15] is given below:

Terminology

Terms used in the implementation are:

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: It is a priority queue containing all active servers, and is arranged according to servers' priorities.

Release-server list: It is a priority queue containing all inactive servers, and is arranged according to servers' activation times. It is used to keep track of the 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 it can be used as feedback to resource management.

Ready-task list: Each server maintains a separate ready-task list to keep track of its ready tasks. It is ordered according to the tasks' priorities. 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 the idling server). In the deferrable server, the idle task of the idle server will run instead.

Data structures

The system maintains two lists: a ready-server list and a release-server list as mentioned earlier. The details of the data structures of these two lists can be found in [15]. The currently executing server in the system is pointed by a running-server pointer. At any time instance, only the tasks of the currently running server can be executed.

Each server within the system contains the `subSystem control block` structure, as depicted in Figure 7.2. It maintains two lists: a ready-task list and a delayed-task list. The delayed-task list is the FreeRTOS list and is used to maintain the tasks when they are not ready (either suspended or delayed) and waiting for their activation.

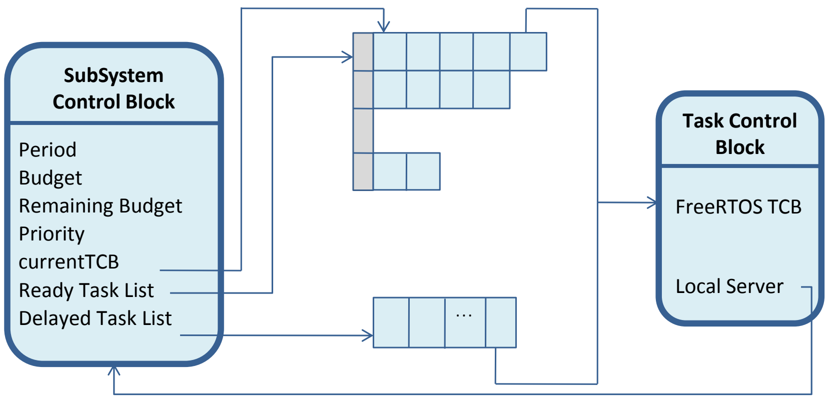


Figure 7.2: Data structures to implement HSF in FreeRTOS

Hierarchical scheduler

The hierarchical scheduling is started by calling `vTaskStartScheduler()` system call and the tasks of the highest-priority ready server starts execution.
Tick-Interrupt handler: At each tick interrupt, the `interrupt_handler` routine performs the following functionality:

- The system tick is incremented.
- Check for the server's activation events. The newly activated servers' budgets are replenished to the maximum values and the servers are moved to the ready-server list.

- The global scheduler is called to handle the server events like execution, activation/replinishment, preemption of lower priority server, suspending the server at budget depletion, etc.
- The local scheduler is called to handle the task events like task execution, activation, preemption of lower priority task, suspension, etc.

Global scheduler: The functionality of the global scheduler is as follows:

- At each tick interrupt, the global scheduler decrements the remaining budget B_s of the running server by one and handles the 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 execute 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 periodic server, the `prvChooseNextIdlingServer()` function selects the first item (with highest priority) from the ready-server list and makes it the current running server. While in the case of a deferrable server, the `prvChooseNextDeferrableServer()` function checks 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 server context-switch is very light-weight, and consists only of changing the running-server pointer from the currently executing server to the newly running server. The ready-task list of the newly running server is activated and all tasks of the list become ready for execution.

Local scheduler: The local scheduler is called from within the tick-interrupt handler routine using an adopted kernel function `vTaskSwitchContext()`. It is the original FreeRTOS scheduler with the following modification:

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.

7.5 Resource sharing in HSF

We implement SRP [12] and HSRP [13, 14] to access local and global shared resources respectively. Since HSRP is an extension of SRP protocol, the SRP terms are extended to implement HSRP and some mechanisms must be implemented to prevent excessive blocking. To use SRP in a hierarchical setup, terms are extended as follows:

- *Preemption level (Priority):* According to SRP, each task τ_i has a static preemption level. Using FPPS, the task's priority ρ_i is used to indicate the preemption level. Similarly, for each subsystem S_s, its priority p_s is used as the preemption level.

- *Resource ceiling:* Each globally shared resource is associated with a *global 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 *local ceiling* for local scheduling. This local ceiling is the highest priority of any task (within the subsystem) using the resource.

- *System and subsystem ceilings:* System and subsystem ceilings are dynamic parameters that change during runtime and the scheduler needs to be extended with the notion of these ceilings. The system ceiling is equal to the currently locked highest global resource ceiling in the system, while the subsystem ceiling is equal to the currently locked highest local resource ceiling in the 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 a 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.

The local and global schedulers are updated with the SRP and HSRP rules respectively and the details are described in Section 7.6.

Now we explain two overrun mechanisms used by HSRP 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 is said to be expired 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 subsystem's budget is replenished at the start of the next subsystem period.

To prevent excessive priority inversion due to global resource lock, it is desirable to prevent subsystem rescheduling during critical sections of global resources. In this paper, we employ the overrun strategy to prevent such rescheduling. According to the overrun concept, upon the budget expiration of a subsystem while its task τ_i has still locked a global resource, the task τ_i is allowed to continue (overrun) its execution until either it releases the locked resource or its overrun time becomes equal to its subsystem budget. The extra time needed to execute after the budget expiration is denoted as *overrun time* θ. We implement two different overrun mechanisms [14]:

1. A 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*: when an overrun happens, the subsystem S_s pays back this consumed amount of overrun in its next execution instant, i.e., the subsystem's budget Q_s will be decreased by θ_s i.e. $(Q_s - \theta_s)$ for the subsystem's execution instant following the overrun (note that only the instant following the overrun is affected since $\theta_s \leq Q_s$).

7.6 Support for resource sharing in HSF

Here we describe the design and implementation details of the resource sharing in two-level HSF. SRP and HSRP are implemented for local and global resource sharing respectively along with overrun mechanisms. The macro `configGLOBAL_SRP` in the configuration file is used to activate the resource sharing. The type of overrun can be selected by setting the macro `configOVERRUN_PROTOCOL_MODE` to either `OVERRUN_WITHOUT_PAYBACK` or `OVERRUN_PAYBACK`.

7.6.1 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 the list has the highest resource ceiling, and represents the `SubSystemCeiling`.

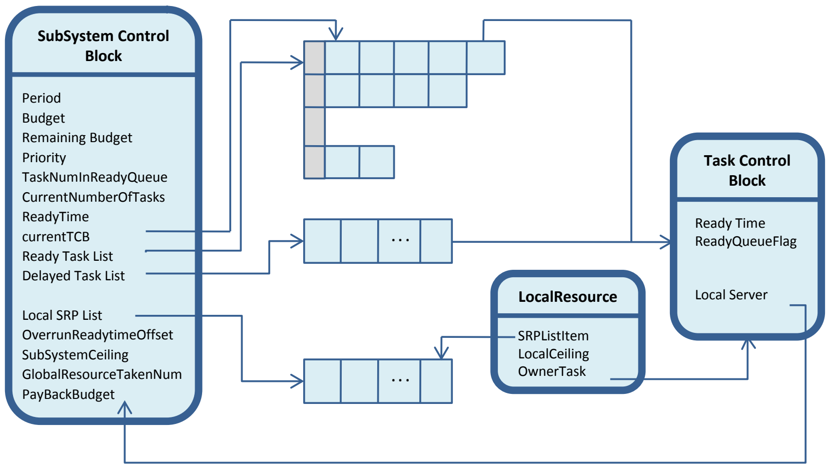


Figure 7.3: Data structures to implement SRP in HSF-enabled FreeRTOS

The extended functionality of the local scheduler for SRP: The only functionality we 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 `SubSystemCeiling`, 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.

7.6.2 Support for HSRP

HSRP is implemented to support global resource sharing among servers. 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 ceil-

ing 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 `SystemCeiling`.

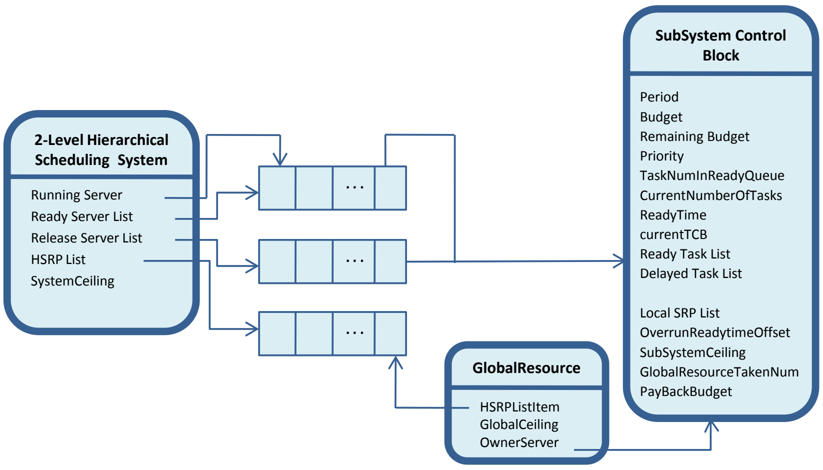


Figure 7.4: Data structures to implement HSRP

The extended functionality of the global scheduler for HSRP: To incorporate HSRP into the global scheduler, `prvChooseNextIdlingServer()` and `prvChooseNextDeferrableServer()` private system calls 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 `SystemCeiling`, 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.

7.6.3 Managing local and/or global system ceilings

To ensure the correct access of shared resources at both local and global levels, the local and global ceilings should be updated properly upon the locking and unlocking of those resources. This functionality is implemented at both local and global levels within the SRP and HSRP API respectively, and is used to lock and unlock the local and global 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` respectively. Moreover, the `SubSystemCeiling` /`SystemCeiling` is updated to the currently locked highest `LocalCeiling` / `GlobalCeiling` of the resource mutex respectively, and the task/server becomes the owner of the local/global resource accordingly. Each time a global resource is locked, the `GlobalResourceTakenNum` is also incremented.

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

7.6.4 Support for overrun protocols

To implement overrun mechanisms in order to prevent excessive priority inversion, the server should continue its execution even if its budget depletes 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.

We have implemented two types of overrun mechanisms; (i) without payback (BO) and (ii) with payback (PO). The implementation of BO is very simple, the server simply executes and overruns its budget until it releases the shared resource, and no further action is required. For PO, we need to measure the overrun amount of time in order to pay it back at the server's next activation.

The data for overrun mechanisms: The `GlobalResourceTakenNum` is used as an overrun flag. As mentioned earlier, it is incremented and decremented at the global resource locking and unlocking respectively. 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 `SubSystemCeiling`) can preempt this server, even if its budget depletes.

Two variables `PayBackBudget` and `OverrunReadytimeOffset` are added to the subsystem structure in order 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`.

The extended functionality of the global scheduler for overrun: A new system call `prvOverrunAdjustServerNextReadyTime(*pxServer)` is used to embed overrun functionality into the global scheduler. For PO, the amount of overrun, i.e. `PayBackBudget` θ_s is deducted from the server `RemainingBudget` B_s at the next activation period of the server, i.e. $B_s = Q_s - \theta_s$.

7.7 Support for legacy application and wrappers

Here we describe the design and implementation details of the legacy server and wrappers.

7.7.1 Creating the legacy server

To utilize the legacy support, a macro `configHIERARCHICAL_LEGACY` must be set in the configuration file `FreeRTOSConfig.h`. The user should rename the old `main()` function, and remove any call to `vTaskStartScheduler()` from the legacy code.

The legacy application is executed in a separate legacy server. The user provides the server parameters like period, budget, and priority for the legacy server. The user also provides a function pointer to the legacy code (the old main function that has been renamed). The `xLegacyServerCreate(xPeriod, xBudget, uxPriority, *pxLegacyServerHandle, *pfLegacyFunc)` system call is provided for this purpose, where `*pfLegacyFunc` is a function pointer pointing to the old main function of legacy application.

The legacy tasks are dynamically attached to the server. The `xLegacyServer-Create()` function first creates a server by calling `xServerCreate(xPeriod, xBudget, uxPriority, *pxLegacyServerHandle)` function. Second, it creates a highest priority private (hidden from the user) task called `vLegacyTask (*pfLegacyFunc)` within the legacy server using the `xServerTaskCreate ( vLegacyTask, pcName, usStackDepth, (void *) pfLegacyFunc, config MAXPRIORITIES - 1, NULL, *pxLegacyServerHandle)` system call.

`vLegacyTask` function executes only once and its main functionality is 1) initializing legacy code (execution of the old main function which creates the initial set of tasks for the legacy application), 2) assigning legacy tasks to the legacy server, and 3) destroying itself; as presented in the Figure 7.5. When the legacy server is executed for the first time, all legacy tasks are created dynamically within the currently running legacy server and start execution.

We have adopted the original FreeRTOS `xTaskCreate` function and developed a wrapper to implement legacy support.

```
// Legacy task function
// function called from xLegacyServerCreate()
static void vLegacyTask (void * pfLF)
{
   ((pfLegacyFunc)pfLF)();  //Initializes legacy code
   vTaskDelete(NULL);       //Destroys itself
}
```

Figure 7.5: Pseudo-code for legacy task implementation

7.7.2 Wrapping FreeRTOS API

Our wrapper implementation consists of repackaging source code interfaces, hence there is no need for modifying the legacy code as depicted in Figure 7.6. The modified functionality is added as *Extended API*, while the original API is kept intact as *FreeRTOS API*. The wrapper provides links to both types of API, and depending on the configuration of `configHIERARCHICAL_LEGACY`, either the modified code or the original FreeRTOS code is executed. This process facilitates the execution of legacy application within the hierarchical environment without making any significant modification in the code.

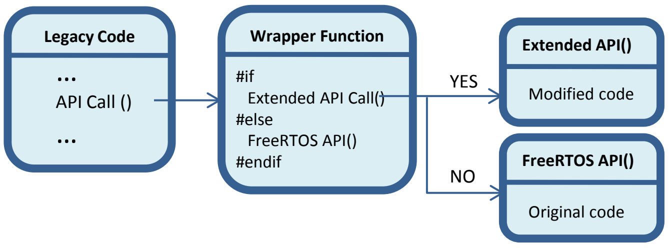


Figure 7.6: Wrappers implementation

Wrapping FreeRTOS API is done in three steps: first, wrappers are constructed, secondly, the original system calls are adapted, and thirdly, the interaction between the wrapper and the legacy programs is tested. The wrappers descriptions are provided in this section, while their testing is performed in Section 7.9.

Wrapper for the legacy task creation

A wrapper is provided for the `xTaskCreate` system call, which redirects the task creation functionality to the `xServerTaskCreate` function by passing an additional parameter of legacy server handle `pxCurrentTCB->pxServer`. This is used to create legacy tasks within the currently executing legacy server, instead of executing the original code of `xTaskCreate` function as shown in Figure 7.7.

```
// Wrapper function for xTaskCreate() system call
xTaskCreate (pxTaskCode, pcName, usStackDepth,
*pParameters, uxPriority, *xTaskHandle, *pStackBuffer,
xMemoryRegion, xRegions)
{
#if (configHIERARCHICAL_SCHEDULING == 1)
   #if (configHIERARCHICAL_LEGACY == 1)
      return xServerTaskCreate (pxTaskCode,
      pcName, usStackDepth, pvParameters, uxPriority,
      pxCreatedTask, pxCurrentTCB->pxServer, puxStack
      Buffer, xRegions);
   #endif
   return pdFALSE;
#endif
// original FreeRTOS code of xTaskGenericCreate
}
```

Figure 7.7: Pseudo-code of wrapper implementation for xTaskCreate API

Wrappers for resource sharing API

To handle synchronization among tasks of a legacy server in HSF and to meet the third challenge, we support the existing FreeRTOS resource sharing API with wrappers. We encapsulate each legacy shared resource within the legacy server by attaching an `owner` server to it, as shown by the newly designed structure `xLegacyQueue` in Figure 7.8. The pointer `pvQueue` points to the original FreeRTOS structure `xQueue`. The definition of semaphore handle `xQueueHandle` is modified to the new structure, as explained by the pseudo-code in Figure 7.9.

An example: To use the wrappers, no change is made in the code of legacy application. For example `xSemaphoreCreateMutex()` is used to create a mutex, which internally calls the `xQueueCreate` function. This func-

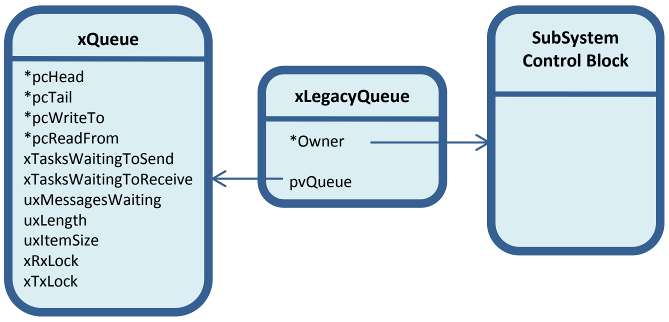


Figure 7.8: Data structures to implement wrappers for FreeRTOS resource sharing API

```
#if (configHIERARCHICAL_LEGACY == 1)
      typedef xLegacyQueue *xQueueHandle
#else
      typedef xQUEUE *xQueueHandle
#endif
```

Figure 7.9: Pseudo-code for the new definition of semaphore handle

tion creates either `xLegacyQueue` or `xQueue` structure, and uses either wrapper code or the original FreeRTOS code depending on the configuration of `configHIERARCHICAL_LEGACY`.

Wrappers for Hardware drivers

A hardware device is accessed by using a hardware driver. The driver provides a software interface to hardware device and it is hardware dependent. A hardware device is usually considered as a global resource that can be shared among tasks of any application. The newly developed applications for HSF can take advantage of using our HSRP protocol implementation for global resource sharing, but the legacy application is not using the newly developed HSRP API. Sharing hardware resources among the legacy and the newly developed applications is a challenge. One simple method is to add the HSRP protocol in the hardware driver by developing wrappers for device drivers as shown in Figure 7.10.

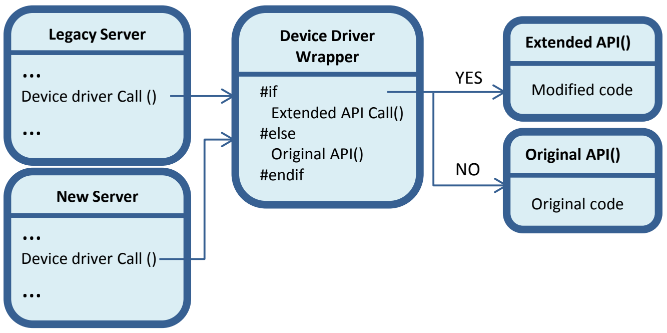


Figure 7.10: Wrappers implementation structure

The `Call()` function in the `extended API` in Figure 7.10 calls the modified code that uses the HSRP protocol within it. USART (universal synchronous/ asynchronous receiver/transmitter) is a highly used driver in embedded systems to send and receive data to and from the embedded device. As an example we present our test results of using wrappers for USART in Section 7.9.2.

7.8 Case study: The legacy applications

In this section we present two legacy applications which are originally developed as stand-alone applications. For both applications, we use the task set presented in Table 7.1, except the resource sharing API which is different. The first legacy application develops a system that endures a priority inversion problem where a high priority task is delayed by the execution of a lower priority task which is not sharing any resource. Second legacy application executes the same code but uses the PIP protocol that solves the priority inversion problem.

The purpose of selecting such a legacy application is to evaluate the wrappers that we have developed for the resource sharing. These legacy applications are very suitable for such an evaluation because they use different resource sharing API (with and without PIP protocol). The intention is to preserve the behavior of both applications, when executed within a two-level hierarchical setup along with other applications.

7.8.1 The legacy applications' design

Each legacy application contains three tasks `TaskL`, `TaskM` and `TaskH` with priorities low, medium, and high respectively, as described in Table 7.1. Note that a higher number in the priority row in Table 7.1 means a higher priority for the task. A resource is shared between `TaskL` and `TaskH`. In `Execution Time` row of Table 7.1, cs represents the execution time of the task within the critical section and the other number shows task execution outside the critical section, e.g. $2cs+1$ means first 2 time-units inside and then 1 time-unit outside the critical section, ($3 + 4cs$ means first 3 time-unit outside and then 4 time-units inside the critical section). The time unit is given in *system tick* which is equal to $1ms$ in our configuration.

`Tasks`	`TaskL`	`TaskM`	`TaskH`
`Priority`	1	2	3
`Period`	120	120	120
`Execution Time`	$2cs+1$	6	$3+4cs$

Table 7.1: Legacy Tasks' properties.

7.8.2 The execution of legacy application in FreeRTOS

Both applications are executed as standalone applications on FreeRTOS using an EVK1100 board. Figure 7.11 provides the pseudo-code of three tasks for both applications. For the first application the shared resource is locked and unlocked using binary semaphore (i.e. for `lock resource R;` and `unlock resource R;` in Figure 7.11), while for the second application it is locked and unlocked using mutex.

Figure 7.12 shows the execution-traces of these tasks as two standalone applications. The left part of the figure demonstrates the execution of tasks using FreeRTOS binary semaphore API and suffering from priority inversion. At tick 6, `TaskH` requests for the shared resource and gets blocked since `TaskL` is accessing the resource. At this point of time, the medium priority `TaskM` executes, thus delaying the highest priority `TaskH` even it is not sharing any resource.

The right part of Figure 7.12 demonstrates the execution of a second Legacy application with the same code but using mutex instead of semaphores. It is obvious from Figure 7.12 (right part) that now the medium priority `TaskM` does not delay the execution of `TaskH`, thereby the priority inversion problem

```
// TaskH function body
// High priority task sharing resource
while (1) {
  execute for 1 tick;
  vTaskDelay(1);           //sleeps for 1 tick
  execute for 2 ticks;
  lock resource R;         //bin.  semaphore or mutex
      execute for 4 ticks;
  unlock resource R;       //bin.  semaphore or mutex
  vTaskWaitforNextPeriod(120);
}
```

```
// TaskM function body
// Medium priority task not sharing resource
while (1) {
  vTaskDelay(1);
  execute for 6 ticks;
  vTaskWaitforNextPeriod(120);
}
```

```
// TaskL function body
// Low priority task sharing resource
while (1) {
  lock resource R;         //bin.  semaphore or mutex
      execute for 2 ticks;
  unlock resource R;       //bin.  semaphore or mutex
  execute for 1 tick;
  vTaskWaitforNextPeriod(120);
}
```

Figure 7.11: Pseudo-code of TaskH, TaskM, and TaskL used for both applications

has been solved by using PIP protocol. Note that our goal is to demonstrate that the behaviour of legacy applications has been preserved when executed in a server within a hierarchical setup and is shown in the next section.

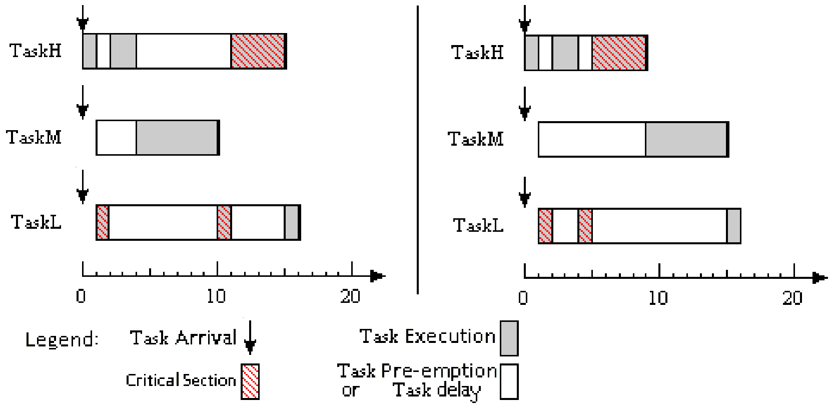


Figure 7.12: The behaviour of both legacy applications: using binary semaphores (left) and mutex (right)

7.9 Experimental evaluation - Results and analysis

This section presents the evaluation of behavior and performance of our legacy server, wrappers, and resource sharing protocols'(SRP, HSRP) implementations. Overheads to execute the newly developed API and wrappers are also measured.

7.9.1 Experiment setup

All experiments are performed on an AVR-based 32-bit `EVK1100` board [17]. The `AVR32UC3A0512` micro-controller runs at the frequency of `12MHz`. The HSF-enabled FreeRTOS is executed on the micro-controller using FPPS policy at both levels for idling periodic servers. The scheduler resolution (system tick) is set to `1ms` (milli seconds).

Four servers are created to perform the behaviour testing. A legacy server named as *`LegacyS`* is created to execute the legacy application. Two servers `S1` and `S2` are used in the system. Additionally, 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. Note that higher number means higher priority for both servers and tasks.

The priorities, periods and budgets for these servers are given in Table 7.2.

Server	S1	S2	LegacyS
Priority	2	3	1
Period	40	20	60
Budget	15	5	10

Table 7.2: Servers used to test system behavior.

Our implementation supports both idling periodic and deferrable servers, however, in this paper we present results with only idling periodic servers. An *idle task* per server is also generated automatically with the lowest priority. It runs when its server has budget remaining but none of its task is ready to execute.

7.9.2 Behaviour testing

The purpose of these tests is to study:

1. the creation, behaviour, and correctness of the legacy server and the legacy tasks.
2. the correct behaviour of wrappers for FreeRTOS resource sharing API, i.e. semaphores, mutexes, etc.
3. the behaviour of global resource sharing using the HSRP protocol:
   (a) between two new servers.
   (b) between the legacy server and a new server.

Tasks	NT1	NT2	NT3	TaskL	TaskM	TaskH
Server	S1	S1	S2	LegacyS	LegacyS	LegacyS
Priority	1	2	1	1	2	3
Period	40	30	60	120	120	120
Execution Time	3	$3cs1 + 1$	$2 + 5cs1$	$2cs2 + 1$	6	$3 + 4cs2$

Table 7.3: Tasks properties and their assignment to servers.

Four experiments are performed to test different behaviours of the implementation. First, the legacy application using FreeRTOS semaphore API is realized within the legacy server and is exercised to test the functionality of the legacy server. Second, the legacy application using FreeRTOS mutex API is realized within the legacy server and the results are compared with the first experiment to validate the correctness of the wrappers for FreeRTOS resource sharing API. Third, the global resource sharing between newly developed applications is tested, and finally in the forth experiment, the global resource sharing between the legacy and the new applications is tested using the HSRP protocol.

All experiment are executed on the micro-controller and the execution traces are visualized using the Grasp tool [24]. The experimental results are presented in the form of visualization of execution-traces in Figures 7.13, 7.14 and 7.15. In these traces, the horizontal axis represents the time in ms, starting from 0. In the server's visualization, numbers along the vertical axis are the server's capacity, the diagonal line represents server execution while the horizontal line represents either the waiting time for the next activation (when budget has depleted) or the waiting time for its turn to execute (when some other server is executing). Since these are idling periodic servers, all the servers in the system execute until their budget is depleted. If no task is ready then the idle task of that server executes till its budget is depleted.

Testing the execution of legacy application in a hierarchical setup

To test the execution of legacy application along with other servers in the system, the first legacy application that endures priority inversion is executed with the previously described servers in Table 7.2. Task properties and their assignments to the servers are given in Table 7.3.

Figure 7.13 visualizes the execution of legacy application within the legacy server along with other servers in the system. The `vLegacyTask` is created within `LegacyS`. It executes at the start of the server only once (at time 25 in Figure 7.13), and creates all other legacy tasks (i.e. `TaskL`, `TaskM`, `TaskH`, and an idle task), assigns them to `LegacyS` and destroys itself. From time 30, the legacy tasks start execution until the server depletes at time 35. The tasks start their execution again, when the server is replenished with its full budget at its next activation period.

Hence, by using the legacy server and a private `vLegacyTask`, the legacy tasks are automatically created and executed within `LegacyS` along with other servers in a hierarchical setup.

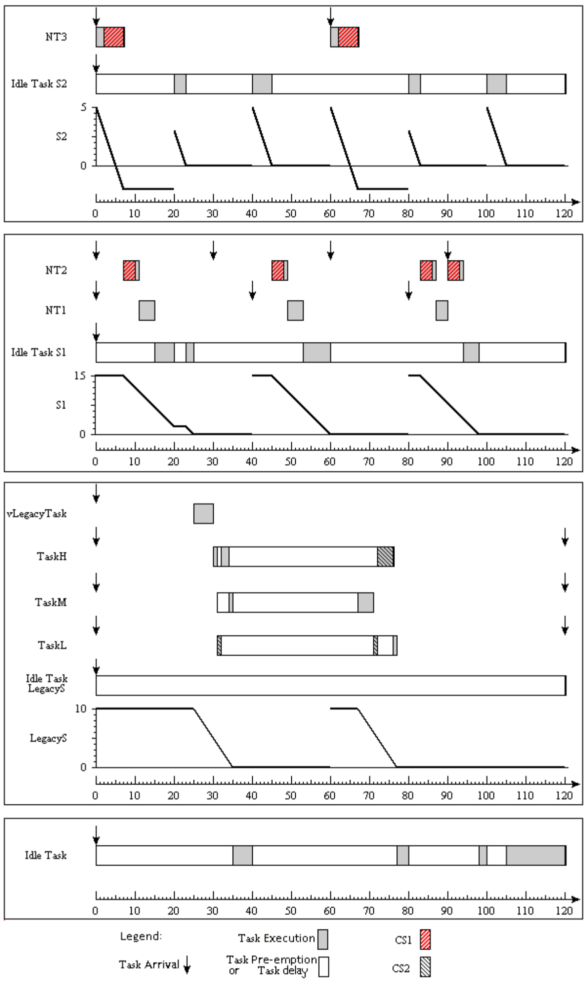


Figure 7.13: The trace for the execution of legacy application within a legacy server using binary semaphores

Testing wrappers for FreeRTOS API by executing legacy tasks within the legacy server

Here the main focus is to test the behaviour of newly developed wrappers for FreeRTOS API.

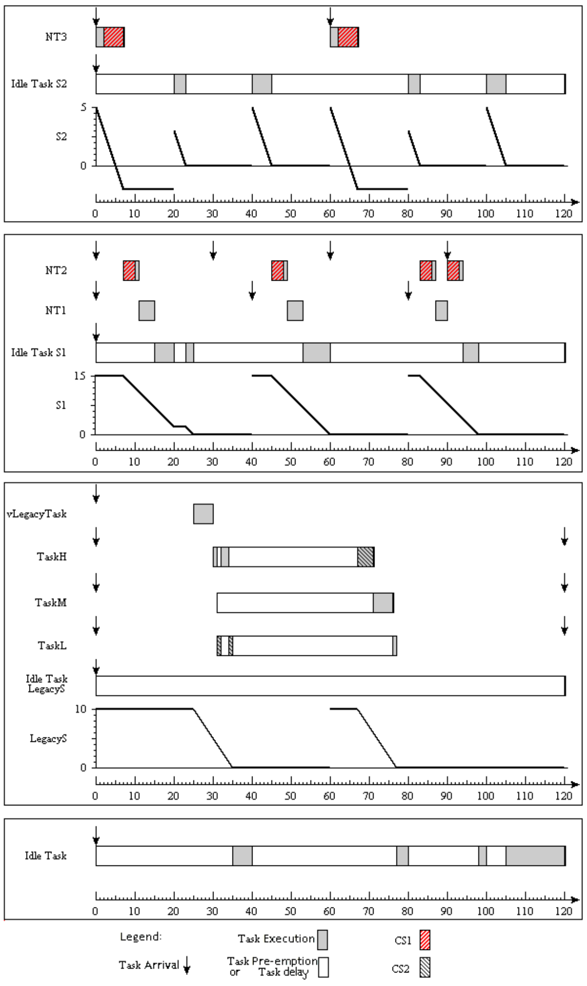


Figure 7.14: Trace showing the legacy server execution using mutex

To perform this test, the first and second legacy applications using FreeRTOS semaphores and mutex API respectively are executed. Servers and tasks properties are provided in Table 7.2 and Table 7.3 respectively. The execution of both applications exhibit priority inversion problems with binary semaphore and its solution with mutex is visualized and presented in Figure 7.13 and Figure 7.14 respectively.

Tasks	NT1	NT2	NT3	LT1	LT2
Server	S1	S1	S2	LegacyS	LegacyS
Priority	1	2	1	1	2
Period	40	30	60	120	120
WCET	4	5	$2 + 5cs1$	$4cs1 + 4$	5

Table 7.4: Tasks properties and their assignment to servers.

`TaskL` and `TaskH` of both applications share a resource, which now becomes a local resource in the hierarchical setup as it is shared among tasks of the legacy server only. The wrappers are executed instead of the original semaphore and mutex API for resource sharing and the critical section is specified by $cs2$. From Figure 7.13 it is obvious that the legacy application suffers from priority inversion, as the TaskH's execution is delayed by the TaskM's execution which is not sharing any resource.

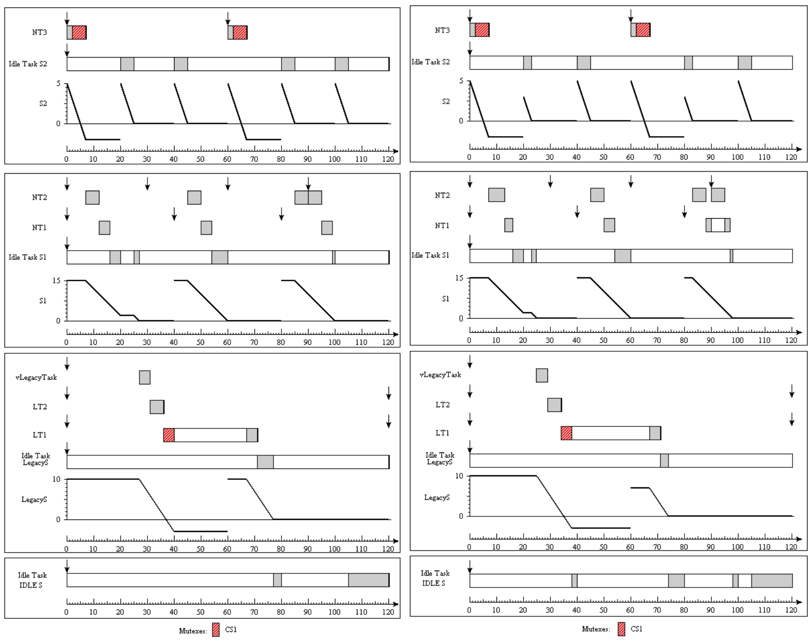
The solution of priority inversion using the mutex API in `TaskL` and `TaskH` is demonstrated in Figure 7.14. Since the mutex implements PIP within them; therefore, the priority inversion problem is solved now. As obvious from the figure that `TaskM` executes after `TaskH`'s completion.

This test shows that the FreeRTOS API is kept intact. It also shows that the legacy application retains its original semantics while executing wrappers for the API system calls in a hierarchical environment. Moreover, the legacy server does not overrun to prevent excessive blocking in both figures since it is not accessing a global shared resource and is not executing newly implemented HSRP protocol.

Testing the global resource sharing between new servers

In this section we test the behaviour of HSRP and overrun in the case of global resource sharing in the HSF implementation. We consider the same servers and tasks as used in the previous tests and which are provided in Tables 7.2 and 7.3 respectively. The trace of execution is visualized in Figure 7.13. Two tasks `NT2` and `NT3`, belonging to servers `S1` and `S2` respectively, are sharing a global resource. The overrun with the payback mechanism is assumed.

In Figure 7.13, `S2` depletes its budget at time 5, but continues to execute in its critical section until it unlocks the global resource at time 7, hence delaying



(a) Trace of budget overrun without payback (BO)

(b) Trace of budget overrun with payback (PO)

Figure 7.15: Testing the behaviour of HSRP and budget overrun between the legacy application and a new server

the execution of `S1` by $2ms$. In case of an overrun with payback, the overrun time is deducted from the budget at the next server activation, as shown in Figure 7.13. At time 20 the server `S2` is replenished with a reduced budget, i.e 3. While in case of an overrun without payback, the server will be always replenished with its full budget.

Testing the modified hardware-driver API

The purpose of this experiment is to test the behaviour of HSRP in the case of global resource sharing between a legacy task and a new task. We are testing the modified hardware driver in which the resource is locked using HSRP. A legacy task `TaskL` of the legacy server shares a global hardware resource

USART with a new task `NT3` of server `S2`. Three servers, as described in Table 7.2, are used in the system. Task properties and their assignments to the servers are given in Table 7.4. The critical section for resource sharing is specified as $cs1$ and the results are visualized in Figure 7.15. The visualization of the executions for budget overrun without payback (BO) and with payback (PO) for idling periodic server are presented in Figure 7.15(a) and Figure 7.15(b) respectively.

In case of budget overrun with payback, the overrun time is deducted from the budget at the next server activation, as shown in Figure 7.15(b). Since HSRP is used, the legacy server overruns at time 35, and later at time 60. The legacy server is replenished with a reduced budget, while in case of an overrun without payback the server is always replenished with its full budget as it is obvious from Figure 7.15(a). It is observed that a hardware resource (USART) is successfully shared among the legacy application and the newly developed HSF applications, without making any modification to the legacy code.

7.9.3 Performance measures

We present the overhead measurements for the wrappers used in legacy application and newly developed resource sharing API for shared resources. A second hardware timer-unit for the micro-controller is initiated and started to measure the performance. The system calls `StartTimer()` and `EndTimer()` are developed to measure execution time of different functions. For each data point, a total of 2000 values are measured. The minimum, maximum, and average of these values are calculated and presented for all results. All data points are given in micro-seconds (μs). The following overheads are measured:

1. The time required to run the wrappers in the hierarchical setup needed to be measured and compared against the original FreeRTOS API to calculate the overhead. The overhead measurements for the semaphore API are given in Table7.5.

2. Similarly, the overhead of executing the modified hardware driver for USART is measured and compared with the original driver. Additionally, we have also tested the effect of passing a different number of characters to the USART driver for printing. The overhead measures are given in Table7.6.

3. We also report the performance measures of lock and unlock functions for the newly developed API supporting SRP and HSRP protocols for

shared both global and local resources. The execution time of functions to lock and unlock global and local resources is presented in Table 7.7.

Function	OS	Min.	Max.	Avg.
`xSemaphoreTake()`	wrapper	22	32	28.47
`xSemaphoreTake()`	FreeRTOS	21	32	26.32
`xSemaphoreGive()`	wrapper	21	32	26.05
`xSemaphoreGive()`	FreeRTOS	21	22	21.51

Table 7.5: The execution time (in micro-seconds μs) of for Semaphore.

Function	Description	Min.	Max.	Avg.
`usart_write_line()`	with global resource sharing	43	54	52.49
`usart_write_line()`	without resource sharing	0	11	9.94

Table 7.6: The execution time (in micro-seconds μs) of USART driver.

Function	Min.	Max.	Avg.
`vGlobalResourceLock`	21	21	21
`vGlobalResourceUnlock`	32	32	32
`vLocalResourceLock`	21	32	26.48
`vLocalResourceUnlock`	21	21	21

Table 7.7: The execution time (in micro-seconds μs) of newly developed global and local lock and unlock function.

The overheads for the semaphore wrappers are very low and negligible, i.e. approximately in average 2 μs for `xSemaphoreTake()` and 5 μs for `xSemaphoreGive()` as it is obvious from Table 7.5. For the hardware driver, we measured the time by passing no character to the USART to exactly measure the overhead as compared by using the driver API. The overhead is approximately 42.55 μs. Additionally, the performance of the USART driver with a varying number of characters is also measured and the results reveal that the increase in the time to execute the code is linear with the increase in the number of characters.

For the server overheads we have performed evaluations in [15] and the results reveal that the overhead measures are low.

7.10 Related work

7.10.1 Consolidating legacy applications

Different types of virtualization techniques are proposed to integrate and execute concurrently multiple applications (including the legacy applications) on a same hardware node using several virtual machines (VMs) and a hypervisor [4]. Examples are without modifying OS [25, 5, 26], or with modifying OS, e.g. Xen-based solution [27, 28]. We focus to execute applications on resource constraint small microcontroller (a 32-bit board), thus executing multiple operating systems is unfeasible and the performance overhead introduced by virtualization/hypervisor layer is a big challenge for such microcontrollers.

OS virtualization or HSF is more lightweight than other virtualizations because of having only a single copy of OS, thus better suited for resource constraint hardware. The hierarchical scheduling processor models guarantee that applications are developed and analyzed independently in isolation and are later integrated together by providing temporal isolation among applications [29, 30, 31, 8, 10]. These advantages make HSF suitable for integrating and executing legacy applications (developed to use the full CPU-access) with other applications (developed to execute in hierarchical setup). However, it requires a priori knowledge of legacy application's timing requirements which has been addressed for independent tasks by [11].

A lot work has been done from the HSF implementation perspective [32, 33, 34, 35, 36] on Linux/RK, open source ERIKA Enterprise kernel, SPIRIT-μKernel, VxWorks, μC/OS-II respectively.Although the reuse of legacy application is proposed by the hierarchical scheduling theoretical work, all mentioned implementations have not proposed special support for facilitating the reuse of real-time legacy application which is the main focus of this paper. To the best of our knowledge, our work is the first to identify challenges and implementation issues and to support a practical implementation for legacy code execution within a server in HSF. No other HSF implementation has investigated on this issue before. Next, we present an overview of the existing synchronization protocols and their implementations in HSF.

7.10.2 Synchronization protocols

Resource sharing for single-level scheduling

Here we describe synchronization protocols used to share resources among tasks in a single-level scheduling systems. Priority inheritance protocol (PIP)

[21] 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) [21] to solve these problems. A slightly different alternative to PCP is the immediate inheritance protocol (IIP). In IIP, the locking task raises its priority to the ceiling priority of the resource, when it locks a resource as compared to the PCP where the locking task raises its priority when another task tries to lock the same resource. Baker presented the stack resource policy (SRP) [12] 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 implement SRP for local-level resource sharing in HSF.

Resource sharing for two-level hierarchical scheduling

To perform independent analysis for applications integration, information about tasks accessing which global shared resources should be known. In a two-level hierarchical scheduling, the resource sharing of a global resource requires to consider the priority inversion at both levels of hierarchy, i.e. between applications at the global level and between tasks within the application at the local level. Multiple synchronization protocols based on SRP [12] have been proposed to accommodate such resource sharing. Fisher *et al.* proposed Bounded delay Resource Open Environment (BROE) protocol [37, 38] for global resource sharing under EDF scheduling. Hierarchical Stack Resource Policy (HSRP) [13] 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) [39] uses the skipping mechanism to avoid the problem of application budget expiration within the critical section. While Rollback Resource Policy (RRP) [40] uses the rollback approach if the budget expires between the critical section. All HSRP, SIRAP, and RRP assume FPPS. The original HSRP [13] does not support the independent application development for its analysis. Behnam *et al.* [14] extended the analysis for the independent development of applications. In this paper we use HSRP [14] for global resource sharing and implement both forms of the overrun mechanism.

Asberg *et al.* [41] implemented overrun and skipping techniques at top of their FPPS HSF implementation for VxWorks and compared the two resource-sharing techniques. Van den Heuvel *et al.* extended the μC/OS-II HSF implementation with resource sharing support [42] by implementing SIRAP and HSRP (with and without payback). They measured and compared the system overheads of both primitives. More recently, Asberg *et al.* [40] implemented and evaluated RRP against HSRP (with and without payback) and SIRAP, and examined that RRP is better in average-case response-times than both protocols.

Unlike [41, 42] and [40] which implement SIRAP, HSRP, and RRP and comparing protocols against each other, we implement HSRP only. We do not consider SIRAP because of its implementation complexity, i.e., worst case execution times of critical sections should be provided during runtime. In addition, we neither consider BROE due to its limitation in supporting FPPS. Our main focus is to enable the reusability of the legacy application and at keeping the semantics of the application intact rather than evaluating different synchronization protocols. To achieve our goals, we keep all FreeRTOS original API intact and call the ones that need to be changed through wrappers implementation.

We aim at efficiency in terms of processor overheads and simplicity in our design with the consideration of minimal modifications in underlying FreeRTOS kernel. Like [36, 35] our implementation limits the interference of inactive servers on system level by deferring the handling of their local events until those servers become active.

7.11 Conclusions and future work

This paper presented the integration and execution of legacy real-time application along with newly developed real-time applications. The focus was to present a solution that pertains the semantics and real-time scheduling properties of old and new applications before and after their integration. We proposed to use the hierarchical scheduling approach (HSF) for this purpose and have demonstrated the suitability of HSF to execute legacy real-time applications in a predictable manner along with other applications. We have identified challenges to execute the legacy application in an HSF setup. Furthermore, we have also described the challenges and implementation issues of enabling resource sharing among the legacy and other applications to make the solution more applicable.

We have presented a runtime support for creating a legacy server and executing the real-time legacy tasks within the server. For resource sharing, we implemented SRP and HSRP protocols for local and global resource sharing respectively. Moreover, to achieve the challenge of resource sharing among legacy and other applications, we have presented the solution by implementing wrappers for the FreeRTOS API.

We have conducted a number of experiments in order to validate the correctness and the efficiency of the proposed solution. We have run our experiments on the EVK1100 board with a 32-bit AVR32UC3A0512 microcontroller. The collected results from the experiments show a smooth execution of legacy tasks integrated with other applications with minimum changes in the code of the legacy tasks. In addition, we could observe, from the experiments, the correct temporal behavior of applications that use our solution when they share software/hardware global/local resources. Finally, the results reveal that the runtime overheads of the proposed solution are rather low. It is done without making any major modification to the legacy code. We have evaluated the implementation of the newly developed API for resource sharing (i.e. SRP and HSRP protocols) and for the wrappers. The results reveal that overhead of our implemented functionality is low.

In the future we plan to extend our solution for multicore architectures, using e.g. the MultiResource Server [43]. It would also be good to perform industrial-scale validation with commercial legacy code.

Acknowledgements

This work is supported by the Knowledge Foundation (KK-Stiftelsen), via the research programme PPMSched.

7.12 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 xLegacyServerCreate(xPeriod, xBudget, uxPriority, *pxLegacyServerHandle, *pfLegacyFunc);`

The user API to implement the local SPR and the global HSPR are the following:

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)`

The new APIs to implement legacy server are the following:

1. `signed portBASE_TYPE xLegacyServerCreate(xPeriod, xBudget, uxPriority, *pxLegacyServerHandle, *pfLegacyFunc);`
2. `signed portBASE_TYPE xServerCreate(xPeriod, xBudget, uxPriority, *pxLegacyServerHandle);`
3. `static void vLegacyTask(*pfLegacyFunc);`
4. `#define xServerTaskCreate(vLegacyTask, pcName, usStackDepth, (void *) pfLegacyFunc, configMAX_PRIORITIES - 1, pxCreatedTask, *pxLegacyServerHandle) xServerTaskGenericCreate((vLegacyTask), (pcName), (usStackDepth), ((void *) pfLegacyFunc), (configMAX_PRIORITIES - 1), (pxCreatedTask), (*pxLegacyServerHandle), (NULL), (NULL))`

Adopted FreeRTOS APIs for wrappers

1. xSemphoreCreateBinary
2. xSemaphoreTake
3. xSemaphoreGive
4. vSemaphoreCreateMutex
5. xSemCreateRecursiveMutex
6. xSemaphoreTakeRecursive
7. xSemaphoreGiveRecursive
8. xSemaphoreCreateCounting
9. xQueueHandle xQueueCreate(unsigned portBASE_TYPE uxQueueLength, unsigned portBASE_TYPE uxItemSize)
10. xQueueHandle xQueueCreateMutex(void)
11. portBASE_TYPE xQueueGiveMutexRecursive(xQueueHandle pxMutex)
12. portBASE_TYPE xQueueTakeMutexRecursive(xQueueHandle pxMutex, portTickType xBlockTime)
13. xQueueHandle xQueueCreateCountingSemaphore(unsigned portBASE-TYPE uxCountValue, unsigned portBASE_TYPE uxInitialCount)
14. signed portBASE_TYPE xQueueGenericSend(xQueueHandle pxQueue, const void * const pvItemToQueue, portTickType xTicksToWait, portBASE_TYPE xCopyPosition)
15. signed portBASE_TYPE xQueueGenericSendFromISR(xQueueHandle pxQueue, const void * const pvItemToQueue, signed portBASE_TYPE *pxHigherPriorityTaskWoken, portBASE_TYPE xCopyPosition)
16. signed portBASE_TYPE xQueueGenericReceive(xQueueHandle pxQueue, void * const pvBuffer, portTickType xTicksToWait, portBASE_TYPE xJustPeeking)
17. signed portBASE_TYPE xQueueReceiveFromISR(xQueueHandle pxQueue, void * const pvBuffer, signed portBASE_TYPE *pxTaskWoken)

Adopted FreeRTOS Private function

1. `signed portBASE_TYPE uxQueueMessagesWaiting(const xQueueHandle pxQueue)`
2. `void vQueueDelete(xQueueHandle pxQueue)`
3. `static void prvUnlockQueue(xQueueHandle pxQueue)`
4. `static signed portBASE_TYPE prvIsQueueEmpty(const xQueueHandle pxQueue)`
5. `signed portBASE_TYPE xQueueIsQueueEmptyFromISR(const xQueueHandle pxQueue)`
6. `static signed portBASE_TYPE prvIsQueueFull(const xQueueHandle pxQueue)`
7. `signed portBASE_TYPE xQueueIsQueueFullFromISR(const xQueueHandle pxQueue)`

Adopted Hardware driver user APIs 1. `void vTaskPriorityInherit(xTaskHandle * const pxMutexHolder)`
The newly added private functions and macros are as follows:

1. `portTickType xServerGetRemainingBudget( void );`
2. `static void prvRemoveGlobalResourceFromList(tskTCB *pxTaskTo-Delete);`
3. `static void prvRemoveLocalResourceFromList(tskTCB *pxTaskTo-Delete);`

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. `OLD void vTaskStartScheduler( void );`

and adopted private functions and macros:

1. `OLD void vTaskSwitchContext( void );`

Bibliography

[1] Charlotte Adams. Product focus: Cots operating systems: Boarding the boeing 787, 2005. [Online]. Available: http://www.aviationtoday.com/, last checked: 20.03.2013.

[2] Charlotte Adams. Reusable software components: Will they save time and money?, 2005. [Online]. Available: http://www.aviationtoday.com/, last checked: 20.03.2013.

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

[4] Z. Gu and Q. Zhao. A state-of-the-art survey on real-time issues in embedded systems virtualization. *Journal of Software Engineering and Applications (JSEA)*, 5:277–290, April 2012.

[5] T. Cucinotta, F. Checconi, and D. Giani. Improving responsiveness for virtualized networking under intensive computing workloads. In *Proceedings of the 13th Real-Time Linux Workshop*, October 2011.

[6] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP' 03)*, pages 164–177, 2003.

[7] Z. Deng and J. W.-S. Liu. Scheduling real-time applications in an open environment. In *Proc. 18th IEEE Real-Time Systems Symposium (RTSS' 97)*, pages 308–319, December 1997.

[8] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proc. 24th IEEE Real-Time Systems Symposium (RTSS' 03)*, pages 2–13, December 2003.

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

[10] L. Abeni, L. Palopoli, C. Scordino, and G. Lipari. Resource reservations for general purpose applications. *IEEE Transactions on Industrial Informatics*, 5(1):12–21, Feb 2009.

[11] L. Palopoli and L. Abeni. Legacy real-time applications in a reservation-based system. *IEEE Transactions on Industrial Informatics*, 5(3):220–228, Aug 2009.

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

[13] 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, December 2006.

[14] M. Behnam, T. Nolte, M. Sjödin, and I. 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.

[15] R. Inam, J. Mäki-Turja, M. Sjödin, S. M. H. Ashjaei, and S. Afshar. Support for Hierarchical Scheduling in FreeRTOS. In *16th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA' 11)*, pages 1–10, 2011.

[16] R. Inam, J. Mäki-Turja, M. Sjödin, and M. Behnam. Hard Real-time Support for Hierarchical Scheduling in FreeRTOS. In *7th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT' 11)*, pages 51–60, July 2011.

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

[18] Richard Stevens, Bill Fenner, Rudoff, and Andrew M. *UNIX Network Programming*. Addison-Wesley, 2003.

[19] OSEK Group. OSEK VDX operating system specification 2.2.3. [Online]. Available: http://www.osek-vdx.org, last checked: 15.05.2014.

[20] Richard Barry. *Using the FreeRTOS Real Time Kernel*. Real Time Engineers Ltd., 2010.

[21] 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.

[22] L. Sha, J.P. Lehoczky, and R. Rajkumar. Solutions for some Practical problems in Prioritised Preemptive Scheduling. In *Proc. 7th IEEE Real-Time Systems Symposium (RTSS' 86)*, pages 181–191, December 1986.

[23] 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.

[24] M. Holenderski, R.J. Bril, and J.J. Lukkien. Grasp: Visualizing the behavior of hierarchical multiprocessor real-time systems. *Journal of Systems Architecture*, 59(6):307–314, June 2013.

[25] T. Cucinotta, D. Giani, D. Faggioli, and F. Checconi. Providing performance guarantees to virtual machines using real-time scheduling. In *Proceedings of Euro-Par Workshops*, pages 657– 664, 2010.

[26] M. Åsberg, T. Nolte, S. Kato, and R. Rajkumar. ExSched: an external cpu scheduler framework for real-time systems. In *18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA' 12)*, pages 240–249, August 2012.

[27] L. Cherkasova, D. Gupta, and A. Vahdat. Comparison of the three CPU schedulers in Xen. *ACM SIGMETRICS Performance Evaluation Review*, 35(2):42–51, 2007.

[28] P. J. Yu, M. Y. Xia, Q. Lin, M. Zhu, S. Gao, Z. W. Qi, K. Chen, and H. B. Guan. Real-time enhancement for Xen hypervisor. In *Proceedings of the 8th International Confer-ence on Embedded and Ubiquitous Computing of the (IEEE/IFIP)*, pages 23–30, December 2010.

[29] 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, September 2004.

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

[31] R. I. Davis and A. Burns. Hierarchical Fixed Priority Pre-emptive Scheduling. In *Proc. 26th IEEE Real-Time Systems Symposium (RTSS' 05)*, pages 389–398, December 2005.

[32] S. Saewong and R. Rajkumar. Hierarchical reservation support in resource kernels. In *Proc. 22th IEEE Real-Time Systems Symposium (RTSS' 01)*, December 2001.

[33] 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.

[34] D. Kim, Y-H 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.

[35] M. Behnam, T. Nolte, I. Shin, M. Åsberg, and R. 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.

[36] M. Holenderski, R. J. Bril, and J. J. Lukkien. *Real-Time Systems, Architecture, Scheduling, and Application*, chapter An Efficient Hierarchical Scheduling Framework for the Automotive Domain. InTech, 2012. ISBN 978-953-51-0510-7.

[37] 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, December 2004.

[38] M. Bertogna, N. Fisher, and S. Baruah. Resource-sharing servers for open environments. *IEEE Transactions on Industrial Informatics*, 5(3):202–219, Aug 2009.

[39] 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.

[40] Mikael Åsberg, Thomas Nolte, and Moris Behnam. Resource sharing using the rollback mechanism in hierarchically scheduled real-time open systems. In *Proc. 19th IEEE Real-Time Technology and Applications Symposium (RTAS' 13)*, pages 129–140, April 2013.

[41] Mikael Åsberg, Moris Behnam, Thomas Nolte, and Reinder J. Bril. Implementation of overrun and skipping in vxworks. In *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'10)*, July 2010.

[42] M. M. H. P. van den Heuvel, R. J. Bril, and J. J. Lukkien. Transparent synchronization protocols for compositional real-time systems. *IEEE Transactions on Industrial Informatics*, 8(2):322–336, May 2012.

[43] R. Inam, N. Mahmud, M. Behnam, T. Nolte, and M. Sjödin. The Multi-Resource Server for predictable execution on multi-core platforms. In *Proc. 20th IEEE Real-Time Technology and Applications Symposium (RTAS' 14)*, pages 1 – 11, April 2014.

Chapter 8

Paper C: Predictable integration and reuse of executable real-time components

Rafia Inam, Jan Carlson, Mikael Sjödin, Jiří Kunčar
In the Journal of Systems and Software (JSS), Vol 91, pages 147-162, May, 2014.

Abstract

We present the concept of runnable virtual node (RVN) as a means to achieve predictable integration and reuse of executable real-time components in embedded systems. A runnable virtual node is a coarse-grained software 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 behaviour is preserved when integrated with other components or when reused in a new environment. Our realization of RVN 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; i.e. deploying functional entities to RVNs and then deploying RVNs to physical nodes, and thus also gives development benefits with respect to composability, system integration, testing, and validation. In addition, we have implemented a server-based inter-RVN communication strategy to not only support the predictable integration and reuse properties of RVNs by keeping the communication code in a separate server, but also increasing the maintainability and flexibility to change the communication code without affecting the timing properties of RVNs. We have applied our approach to a case study, implemented in the ProCom component technology executing on top of a FreeRTOS-based hierarchical scheduling framework and present the results as a proof-of-concept.

keywords: Real-time software components, component integration, component reuse, hierarchical scheduling, delay analysis.

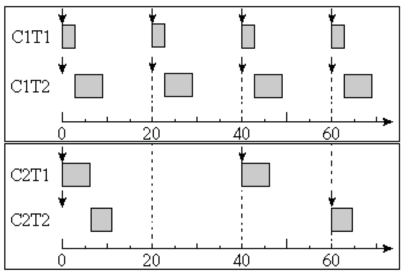
8.1 Introduction

In this paper we target development of the large class of embedded systems which is required to perform multiple simultaneous control-functions with real-time requirements. From the 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.

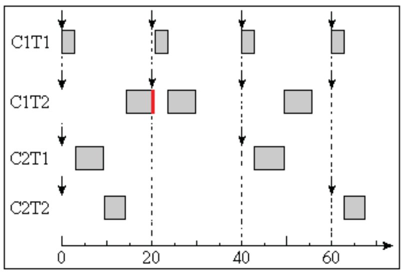
Temporal behavior of real-time software components poses difficulties in their integration. When multiple software components are deployed on the same hardware node, the emerging timing behavior of each of the components is typically unpredictable. For example, the temporal behaviour of two components $C1$ and $C2$ and their tasks execution is depicted in Figure 8.1, where the horizontal axis represents time, an arrow represents task arrival and a filled rectangle shows task execution. The temporal behaviour of both components is tested to be correct during unit testing and all tasks of both components meet their deadlines when executed separately before integration as obvious from Figure 8.1(a). However, upon their integration, tasks of one component affect the scheduling of tasks of other components and as a result task $C1T2$ misses its deadline at time 20 in Figure 8.1(b). This means that for an embedded system with real-time constraints; 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.

In this paper we focus on the schedulability of tasks, i.e. meeting their deadlines, as the main timing property. An RVN's timing behaviour is *predictable* during its integration and reuse, as long as the schedulability of tasks that have been validated during its development within a component is guaranteed when components are integrated together.

In the real-time community, *Hierarchical Scheduling Framework (HSF)* [2] is known as a technique for solving this predictability problem by providing temporal isolation between components. It supports CPU time sharing among components or applications (means leveraging the CPU-time partitioning from



(a) Tasks' execution within separate components



(b) Tasks' execution after components' integration

Figure 8.1: Schedulability problem during components' integration.

task-level to the component-level by executing each component in a separate server), hence isolating components' functionality from each other for, e.g., temporal fault containment, compositional verification, and unit testing. HSF has been proposed to develop complex real-time systems by enabling temporal isolation and predictable integration of software-functions [3].

We address the challenges of preserving the timing properties within components and to apply these properties during components' integration. We propose the concept of a *runnable virtual node (RVN)*, in which 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. An RVN represents the functionality of software-component (or a set of integrated components) combined with allocated timing resources and a real-time scheduler to be executed as a server in the HSF. 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 in *a first-step*, and then, deploying multiple virtual nodes to the physical node (target hardware) in *a second-step*.

An important feature during component integration is to provide communication among various components of a target software system. This communication should also be predictable in case of real-time components and do not affect the schedulability of tasks. We implement a communication strategy that enables to execute the communication code independently from RVNs hence making the RVNs integration predictable, since communication time will not affect the schedulability of RVNs' tasks.

The main contributions of this paper are:

- We *realize the concept of runnable virtual nodes* for the ProCom component technology [4] by exploiting the HSF implementation [5]. The purpose is to make the integration of real-time components predictable, and to ease the component's reuse in the new systems.

- We *introduce a two-level deployment process* instead of a single big deployment. The two-level process gives development benefits with respect to composability, system integration, testing, validation and certification. Further it leverages the hierarchical scheduling to preserve the validity of an RVN's internal temporal behaviour when integrated with other components or when reused in a new environment.

- We *implement a communication strategy* that supports the predictable integration and reuse of runnable entities. We *evaluate this strategy* against a direct strategy for efficiency and reusability aspects of RVN. We *develop an analysis tool End-to-End Latency Analyzer for ProCom (EELAP)* [6, 7] to compute the end-to-end latencies of both communication strategies / or to evaluate both communication strategies.

- We *provide a case study* as a proof-of-concept of our approach: we implement it using the ProCom component technology and execute it on a real hardware an AVR 32-bit microcontroller [8]. We *demonstrate the runnable virtual node's properties* with respect to temporal isolation and reusability.

Once the RVN is assigned for timing properties, it will preserve these properties without regard of other RVNs it is integrated with on the same physical node. Our realization allows predictable coexistence of virtual nodes that have been either constructed with different development methodologies or constructed using the same development technology but having different timing properties. E.g., a ProCom-RVN can co-exist with an RVN with legacy FreeRTOS-tasks, or an RVN with hard real-time components that has been verified with formal methods can co-exist with an RVN with components without real-time requirements and that has not undergone extensive validation.

The RVN discussed in this paper is the extension of our previous work on the concept of virtual node in [9] and the initial implementation in [10]: both papers are based on the idea of a real-time component that preserves its timing properties when integrated with other components on a physical platform. The work described in this paper is the extension of the initial implementation of the concept of the RVN component, and the synthesis of the final executables with the emphasis on using a two-step deployment process. The previous work of [9], on the other hand, focused on just the presentation of the general idea of virtual node as a real-time component at a high level of abstraction and described inclusion of virtual nodes within different components technologies like AUTOSAR, AADL, and ProCom. It did not address the synthesis process and lacked a practical implementation.

The previous work of [10] only presented the initial RVN implementation for components integration, and the evaluation of predictable integration of real-time components using a case study on cruise controller system and its execution in ProCom component model on the AVR-based board EVK1100 [8]. In this paper we extend the implementation to incorporate *the reuse of real-time components along with their timing properties* and discuss how the two-level deployment helps accomplishing the predictable coexistence of real-time components together and facilitate their reuse. We also evaluate the reuse property of RVN by extending the previous case study with the new functionality of the adaptive cruise controller system and presenting new test results to prove RVN as an executable reusable entity.

Compared to [10], another significant extension in this paper is the implementation of an inter-RVN communication strategy that supports predictable integration and reuse of RVNs. In addition we have also implemented the EE-LAP analysis tool [6] and have evaluated the communication strategies using the tool.

Outline: Section 8.2 gives an overview about the ProCom component technology on which our work is based. Section 8.3 describes the RVN concept in details including its deployment process and how it embeds HSF within it. Section 8.4 explains the communication strategies among RVNs, and details of synthesis activities are given in Section 8.5. In Section 8.6, we explain how the two-level deployment process preserves timing properties within RVNs. Section 8.7 describes end-to-end delay analysis and explains end-to-end latency calculations for inter-RVN communication strategies. Section 8.8 presents a case-study in which RVNs are used for the ProCom technology. We present the results of evaluations of (1) preserving timing properties during integration and reuse of RVNs, and (2) end-to-end latencies for both communication strategies in Section 8.9. Section 8.10 presents related work on component-based technologies, and finally, Section 8.11 concludes the article.

8.2 The ProCom component technology

The ProCom component technology targets control-intensive embedded systems like software used in trains, airplanes, cars, and industrial robots, etc. The ProCom component model [4] is specifically developed to address the reuse of design artefacts (e.g., extra-functional properties, analysis results, and behavioral models) as well as predictable integration and reuse of the executable components [11]. The PROGRESS Integrated Development Environment (PRIDE) tool [12] supports modeling and automatic-synthesis of components at different levels [13].

The ProCom component model can be described in two distinct realms: the modeling and the executable realms as shown in Figure 8.2. In the Modeling realm, the models are made using component-based and model-based development while in the executable realm, the synthesis of runnable entities is done from the model entities.

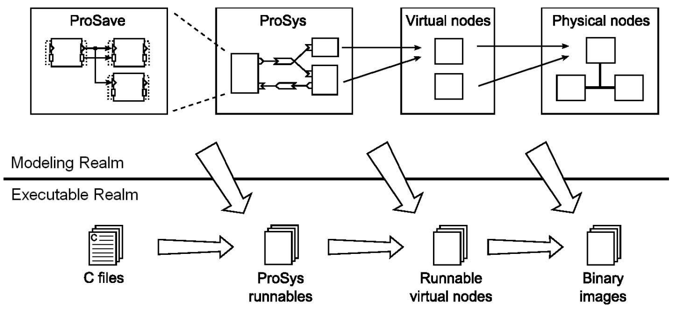


Figure 8.2: An overview of the modeling formalisms and synthesis artefacts.

8.2.1 The modeling realm

Modeling in ProCom is done by four discrete but related formalisms as shown in Figure 8.2. 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 the ProSave component level (passive in nature). At the second formalism level, many ProSave components are mapped to make a complete subsystem called ProSys (active in nature) [4].

The deployment modeling is used to capture the deployment related design decisions and then mapping the system to run on the physical platform. Multiple ProSys components can be mapped together on a virtual node together with a resource budget required by those components. After that, many virtual nodes could be mapped on a physical node. The relationship is again many-to-one. This part represents all the physical nodes and their inter-communication [14].

8.2.2 The executable (or runnable) realm

This realm presents the synthesis of executables/runnables 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 a collection of operating system tasks. RVNs implement the local scheduler and contain the tasks in a server (details are given in Section 8.3.2). Final binaries are generated by connecting different RVNs together with a global scheduler and using a middleware API to provide communications among RVNs.

8.3 Runnable Virtual Node (RVN)

The previous section presented a general background of the ProCom component model. This section focuses on the details of RVN. It explains the RVN concept and its deployment mechanism in ProCom, the internal details of RVN as a server using an HSF implementation in the FreeRTOS operating system in Sections 8.3.1, 8.3.2 respectively.

8.3.1 The RVN concept and its deployment mechanism

A runnable virtual node is an execution platform concept that preserves functional as well as temporal properties of the software executed within it [11]. It is intended for coarse-grained components, for single node deployment, and with potential internal multitasking. The idea is to encapsulate the timing properties into reusable executable components to achieve predictable integrations and reusability of those components along with increased maintainability, testability, and extensibility.

In ProCom, a runnable virtual node is an integrated model concept. It means that the virtual nodes exist both on the modeling and on the executable levels as shown in Figure 8.2.

The two-level deployment process is used to synthesize RVNs and final executables, so the modeling hierarchy is also maintained at run-time by executing the tasks within each RVN, instead of flattening the whole system to a single level of tasks. The two-level deployment process is depicted in Figure 8.3. It preserves the timing properties of each RVN and gives development benefits with respect to composability, system integration, testing, validation, and reuse. More on the timing properties of RVN is explained in Section 8.6.

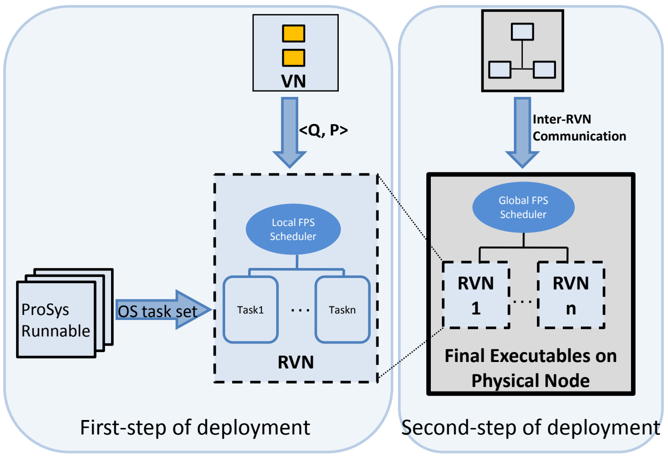


Figure 8.3: The two-level deployment process in ProCom component technology.

From the modeling perspective, a *virtual node (VN)* is a container for a set of integrated ProSys-components plus the execution resources (a budget and period) required for these ProSys-components. The input and output ports of those components are inherited by the virtual node during the first-step of deployment. A *physical node* is modelled by integrating different VN together on a single platform and defining communication among them during the second-step of deployment.

In the executable form, an RVN is constructed by mapping the set of tasks (synthesized from ProSys-runnables) to a server and assigning scheduling parameters (assignment of task-priorities) during the first-step of deployment. It also adds an implementation of inter-RVN communication using message channels to send messages among virtual nodes. The final binaries are generated for a hardware node during the second-step of deployment by connecting different RVNs together with a global scheduler, assigning server-priorities, and using a *middleware API* for *inter-RVN communications*. 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 (we use fixed-priority preemptive scheduling); hence components priority assignment is done during the final step of integrating virtual nodes to physical nodes. All synthesis is done by generating C-code, so the final step is compiling the generated code, and linking all code with the operating system and middleware binaries.

8.3.2 The RVN-server

An RVN is implemented as a server within a two-level HFS (also called an RVN-server), and includes a set of tasks, a resource allocation $\langle Q, P\rangle$ (where P is the server period and Q is the server budget ($0 < Q \leq P$), Q is the portion of CPU time allocated periodically to the server), and a real-time scheduler as shown in Figure 8.4. We follow the periodic resource model [15], our servers and tasks are activated periodically. Servers behave like periodic tasks, they replenish their budget Q every constant period P. In two-level hierarchical scheduling, the CPU time is partitioned among a set of subsystems (or servers), RVN-servers in our case. The RVN-servers are scheduled by a global (system-level) scheduler. Each RVN-server contains its own internal set of tasks that are scheduled by a local (subsystem-level) scheduler.

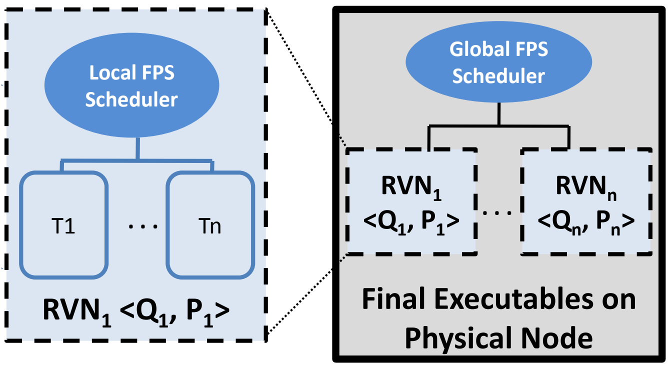


Figure 8.4: An RVN-server within a two-level hierarchical scheduling.

The global scheduler schedules the servers according to their allocated CPU resource, i.e. each server executes for a specified time (budget Q) during each period P. At the budget depletion, the server stops its execution and waits for its next activation period when it replenishes with its full budget. The local scheduler schedules tasks of a server only when the server is executing. Its tasks cannot run while the server is not executing (either due to preemption by a higher priority server, or waiting for its next activation period). Accordingly we say that the HSF provides partitioning of the CPU between different servers [2].

In our implementation, the RVN-server is executed in the two-level HSF running on-top of FreeRTOS [16]. FreeRTOS is a portable open source real-time kernel which is small and scalable, supports over 20 different hardware architectures, and is easy to extend and maintain [17].

The official release of FreeRTOS only supports a single level fixed-priority scheduling. We have, however, previously presented an implementation of two-level HSF for FreeRTOS [5] with associated primitives for hard real-time sharing of resources both within and between servers [18]. The two-level HSF implementation supports two kinds of servers, idling periodic [19] and deferrable servers [20]. In this paper we present our results with the idling server. The implementation uses Fixed-Priority Preemptive Scheduling (FPPS) for both global and local-level scheduling. For local resource-sharing (resources shared among tasks within the same RVN-server) the Stack Resource Policy (SRP) [21] is used, and for global resource-sharing (resources shared among tasks of different RVN-servers) the Hierarchical Stack Resource Policy (HSRP) [22] is implemented. The HSF supports CPU resource reservations by associating a tuple $\langle Q, P \rangle$ to each server. 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 [18].

The HSF gives the potential to (1) develop and analyze subsystems in isolation from each other, (2) execute the server with a guaranteed temporal behaviour regardless of any other execution on the physical node as long as its allocated CPU resource is provided, and (3) reuse the real-time properties of subsystems along with their functional properties [23]. As each RVN-server has its own local scheduler, after satisfying the temporal constraints, its non-functional (timing) properties are preserved within each RVN-server along with its functional properties when the RVN is integrated with other RVNs on a physical node, or when it is reused in another context [11]. Later, a global scheduler is used to schedule all the RVN-servers together according to their resource reservations without violating the temporal constraints that are already analyzed and stored in the RVN-server. Hence as long as the CPU resource allocation $\langle Q, P \rangle$ is providedto an RVN-server, all tasks of the RVN-server will meet their deadlines, no matter with which other RVN-servers they are integrated / executed with on the physical platform.

Thus using HSF, the functionality of different RVN-servers can be isolated from each other for, e.g., fault containment, compositional verification, validation and certification, and unit testing. Further, these RVN-servers can be reused in the new system reusing not only the functionality but also their temporal properties.

8.4 Inter-RVN communication

This section discusses two different the inter-RVN communication mechanisms when multiple RVNs are integrated together on a physical node.

As stated earlier, the RVN provides main benefits of predictable integration and increased reusability of executable real-time components. We focus to develop such an inter-RVN communication that should facilitate integration and reuse of the executable components. Additionally, it should also be fast and predictable. To achieve these benefits, the inter-component or inter-RVN communication is implemented independently from the underlying platform as a *middleware API* by moving the information about system and communication outside the component code. Later the middleware interface functions are integrated into the layered ProCom model [24].

Middleware API: The inter-RVN communication is a combination of data and trigger ports and is based on messages. The middleware API is implemented a-synchronously via message passing and the cyclic shared buffers, where channels are used to distribute the messages to the other RVNs using a defined set of connections. The communication is independent of the underlying operating system (HSF implementation in our case). It includes the support for transparent communication within RVNs mapped on the same hardware node, called *local-RVN communication*, and among RVNs mapped on different hardware nodes through a communication media or channel (e.g. CAN bus), called *distributed-RVN communication*. The final executables are generated by resolving the local-RVN communication by mapping it to middleware API, and synthesizing the distributed-RVN communication among hardware nodes (if needed).

The middleware API is well-integrated with the layered deployment process of the ProCom model. Currently it provides the local-RVN communication where the output and input message ports write and read the data respectively. One-step shared cyclic buffers that can be accessed by multiple tasks, are added to these ports for reliable message delivery and efficient memory usage. The main communication code (including data structures) is initialized and two periodic tasks *sender* and *message-port updater (or receiver)* are created for every physical node during the second-step of deployment. These tasks are responsible to send and receive messages among RVNs respectively. For the distributed-RVN communication, the only additional functionality to be generated is a communication channel. (At this stage the PRIDE tool provides the distributed-RVN communication in the form of a TCP/IP connection over Ethernet. A real-time distributed-RVN communication is not automatically generated by the tool and has to be provided manually).

The inter-RVN communication could be realized in two different ways: either integrating the middleware API directly into the communicating RVNs, called *Direct Communication*, or using a server to embed the middleware tasks in it, called a *Server-based Communication*. Both strategies are explained here.

8.4.1 Direct communication strategy

The simplest and straightforward method to provide communication among RVNs is a direct communication strategy where RVNs can communicate directly with each other (i.e. RVN-to-RVN). Shared message queues which are accessed via the SRP API [18] can be used for this purpose. An RVN can encapsulate the middleware API to send and receive the data and/or messages (see Figure 8.5) at the first-step of deployment. It requires a separate configuration of each RVN for each communication. The final binaries are generated from RVNs along with the code for the communication mechanism used (local- or distributed-RVN) at the final-step of deployment. The budget and period of the communicating RVN includes both the timing requirements to execute RVN code, and the timing requirements to execute the communication code (sender and receiver tasks).

The direct communication is fast as compared to the server-based communication. However, it reduces the reusability of executable RVN components since RVNs need to be configured for each system separately, depending on that system's communication requirements. Further, any change in the middleware communication code will not only require a code-change in all RVNs but will also affect the timing properties of all RVNs involved in that communication.

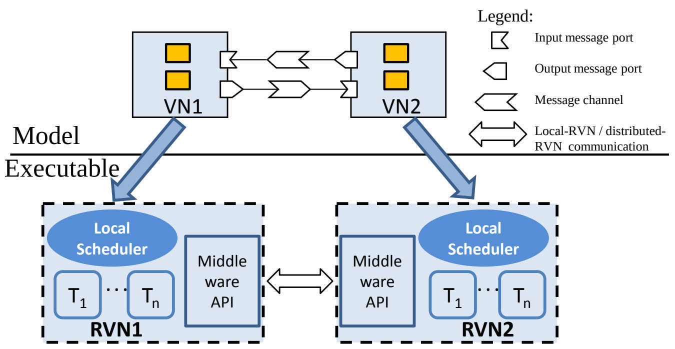


Figure 8.5: Direct inter-RVN communication at modeling and executable levels.

8.4.2 Server-based communication strategy

Since RVNs are implemented as servers within a two-level HSF, it makes sense to embed the middleware API within a separate server called a *system (also called a communication) server*. The main functionality of the server is to send and receive messages among the RVNs, i.e. to copy the messages from the sender port of one component to the receiver port of another component, by executing the middleware API tasks within it. To achieve this purpose, both communication tasks, sender and receiver, are assigned to the server as shown in Figure 8.6. Hence using this strategy, the timing properties of the communicating RVNs (RVN1 and RVN2 in Figure 8.6) become independent of the communication code and the time to execute communication code.

Since inter-RVN communication is implemented as a server within a two-level HSF, simple semaphores cannot be used to protect shared buffers. Thus some synchronization protocols for hierarchical schedulers are required to access the buffers in a safe manner. SRP and HSRP protocols are implemented in the HSF implementation [18]. The shared buffers among the tasks of same RVN and among tasks of different RVNs are arbitrated using SRP and HSRP respectively. HSF leverages the communication among components with the advantages of short and predictable global blocking, and predictable and well-defined communication.

The server-based communication strategy is more complicated than the direct strategy and has additional overhead of system server's execution, however it proposes additional properties/advantages to incorporate/enhance the

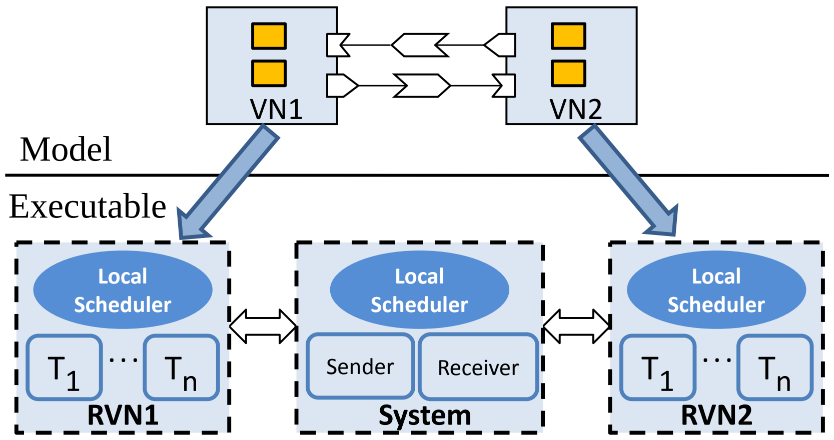


Figure 8.6: Server-based inter-RVN communication at modeling and executable levels.

predictability of RVN during their integration and reuse. Main advantages of the strategy are (1) increased reuse of RVN by keeping the communication separated from the RVN code that increases maintainability and flexibility to change the RVN without affecting the communication (2) predictability by executing the communication API in a server within the HSF implementation. Moreover, it also increases (3) maintainability and flexibility to change the communication code without affecting the timing properties of RVNs.

8.5 Synthesis activities

The automatic synthesis of ProCom components is done at different levels:

8.5.1 Synthesis of ProSave and ProSys runnables

The primitive ProSave components are implemented as C-functions. Synthesis for the second level includes assigning ProSys components to a set of tasks to generate ProSys runnables. Since the tasks at this level are independent of the execution platform, the only task-attributes assigned at this stage are the period for each periodic task (which is taken from the clock frequency that is triggering the specific components in the task) and minimum inter-arrival time for each event-triggered task (which is taken from the model of the events) [13].

8.5.2 Synthesis of RVNs and final executables

Since a virtual node is functionally equivalent to a set of ProSys-components, most of the synthesis work is done at the ProSys-level. For synthesizing the RVN the main issue is to assign priorities to the tasks in order to meet timing constraints on components. In the PRIDE tool, the currently implemented priority assignment is Rate Monotonic.

The final executables are generated by assigning priorities to the servers executing RVNs, resolving the local-RVN communication by mapping it to middleware API (which use the same middleware for communication that the ProSys-components use within an RVN), and synthesizing the distributed-RVN communication among hardware nodes (if needed). All synthesis is done by generating C-code, so the final step is compiling the generated code, compiling the behaviors of the ProSave components, and linking all code with the operating system and middleware binaries.

8.5.3 Synthesis of server-based communication

We have implemented support for server-based inter-RVN communication within the PRIDE tool. The communication or system server along with its timing properties is automatically generated for inter-RVN communications (if needed), at the second-step of deployment. Both communication tasks (sender and receiver) are assigned to the server and the server parameters (period, budget, priority) are automatically generated. The system server has the highest priority among all the servers in the system, with a very small budget to keep the communication overhead as low as possible. Additionally there can be a hardware-driver task in the server if needed.

8.5.4 Synthesis of idle server

An *idle server* is contained within the HSF implementation [5]. 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 servers, i.e. 0. It contains only a single idle task. This is useful for tracing the temporal separation among servers and also useful in testing system behavior. Its presence is also useful in detecting over-reservations of server budgets and can be used by the system to optimize the resource usage.

8.6 Predictability and reusability of RVNs

In this section we explain how the two-level deployment process to synthesize RVN-servers, and inter-RVN communication mechanism help in achieving predictable integration and increased reusability of RVNs.

As stated earlier, the RVN-server within the two-level HSF provides temporal isolation, independent development and analysis, and preserves temporal properties within the server, which leads to predictable integration. By predictable integration we mean that all servers and tasks meet their deadlines as long as the allocated resources are provided to the server. Further, server-based communication makes the RVNs independent of communication (code and time to execute that code), since they do not know at design and execution level about other RVNs they need to communicate with, thus increasing RVN's reusability in the new systems.

The two-level deployment process leverages the hierarchical scheduling to preserve the validity of an RVN's internal temporal behaviour when integrated with other components or when reused in a new environment. During first-step of deployment the timing properties of each RVN are validated and preserved along with its functionality, and during the second-step of deployment RVNs are integrated along with their preserved timing properties, as explained here:

8.6.1 The first step of deployment

In the modeling realm, a virtual node consists of a set of ProSys-components with an added resource reservation $\langle Q, P \rangle$. This resource reservation makes it possible to start reasoning about the timing properties of different components inside the virtual node (i.e. inside the top-level ProSys-components). In the executable realm, the RVN is constructed by mapping the set of tasks that have been synthesized from the integrated ProSys-components to a server and assigning scheduling parameters (which in the current implementation means assignment of task-priorities). Internal validity of the timing-constraints of the RVN can then be assessed using, e.g., simulation, testing or a *local schedulability-analysis* provided in [18]. In this manner, after configuration the RVN-server preserves its timing properties.

8.6.2 The second step of deployment

In the executable realm, we create the final binary for a hardware node by mapping a set of RVNs to that node along with a global-level scheduler in HSF, resolving *local-RVN communications* (communication among RVNs mapped on the same hardware node), and mapping *distributed-RVN communications* (communication among RVNs mapped on different hardware nodes) with remote RVNs to the communications media. At this point it is also necessary to assign scheduling parameters in terms of server-priorities. To ensure that a feasible priority assignment has been made, a *global schedulability-analysis* [18] is performed.

It should be noted that the global schedulability-analysis is done using only information about the allocations ($\langle Q, P\rangle$:s) of RVNs and their mutual global resource-sharing. The extent of the global resource-sharing is known; since the only source of global resource-sharing is the local-RVN communication which has been automatically generated. Thus, our approach does not require to use only RVNs that have been synthesized from ProCom-components (with well-known and analyzable behavior). It is perfectly feasible to wrap any type of component which can be represented as a set of tasks as an RVN and to integrate it with pure ProCom-RVNs. The global scheduling-analysis will guarantee that timing requirements within the RVNs are met at run-time and the HSF will guarantee that execution resources are provided according to the reservation. Thus, for example, a legacy-component with an unanalyzable task-set which has been tested and informally validated with a specific reservation, will continue to work when integrated with other RVNs. Another example would be when a, potentially, unreliable third party component should be integrated. By wrapping such a component as an RVN, it is guaranteed that the component does not interfere adversely by consuming more than its allocated execution resource.

8.7 End-to-End delay analysis and its computation method

In embedded systems' communication, the data may originate at one component (e.g. a sensor) and passes through various other computational components, before terminating at the final component (e.g. an actuator). Hence, the data follows a chain of components $(C1, C2, \ldots, Cn)$, each potentially having its own periodicity and timing properties. The total time taken by the data/signal to traverse the complete chain is called *end-to-end delay* (also called *end-to-end latency*) [25]. For an embedded system with real-time constraints, the end-to-end timing behavior is not only dependent on the timing properties of its constituent components but also on the message-chains among those components. In a communication chain, different executable components (or tasks) are activated at different periods. Such system is called a *multi-rate system* and different end-to-end semantics for the multi-rate system are provided in [26].

The communication strategies for ProCom technology reveal the multi-rate systems. Since RVNs are implemented as servers in the ProCom, a system comprising the communication chains among RVN-servers transposes to a *multi-rate server-based system*.

For hard real-time systems, of course, *Worst Case Response Time (WCRT)* (the largest possible response-time of any instance of the task) is a more interesting metric than any other metric, however, it does not help us to evaluate a multi-rate server-based systems. We need to compute end-to-end delay analysis to evaluate both communication strategies. However, the end-to-end delay analysis of [26] cannot be straightforwardly applied to the multi-rate server-based ProCom components. The WCRT for the communicating tasks executing within the two-level hierarchy of servers should be first computed and then provided as input to compute end-to-end latencies.

To describe the computation methods for WCRT and end-to-end latencies for ProCom components, we first present an Example of communication chains for direct and server-based communication strategies, and then we explain the computations of WCRT and end-to-end latencies for the servers and tasks presented in the Example. In the end of this section, we provide a brief description of the EELAP tool [6, 7].

8.7.1 An example

Consider a system with three RVN-servers, a sensor, a compute, and an actuator as depicted in Figure 8.7. Each server has different timing properties given in Table 8.1, and contain different task sets described in Table 8.2. The data is generated at the task τ_{11} and sent to the task τ_{12} for computation, then the computed data is sent to τ_{13}. The task τ_{22} is not involved in the communication.

RVN-Server	Sensor	Compute	Actuator
Period	25	40	10
Budget	10	4	2
Priority	3	2	1

Table 8.1: Timing properties of servers used to test system behavior.

Tasks	τ_{11}	τ_{12}	τ_{22}	τ_{13}
`Server`	`Sensor`	`Compute`		`Actuator`
`Period`	40	80	120	20
`Exe. Time`	2	2	4	2
`Priority`	2	2	1	2
`WCRT`	32	74	116	18

Table 8.2: Timing properties of tasks in RVN-servers.

The communication chain for the direct strategy is $\tau_{11} \rightarrow \tau_{12} \rightarrow \tau_{13}$ as obvious in Figure 8.7. For server-based strategy, the inclusion of an extra System server in the communication chain results in a longer chain $\tau_{11} \rightarrow$ `sender` $\rightarrow$ `receiver` $\rightarrow \tau_{12} \rightarrow$ `sender` $\rightarrow$ `receiver` $\rightarrow \tau_{13}$ as given in Figure 8.8.

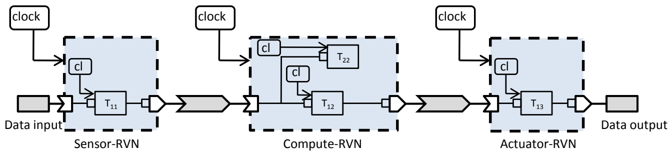


Figure 8.7: A Communication Chain among RVN-servers for Direct Communication.

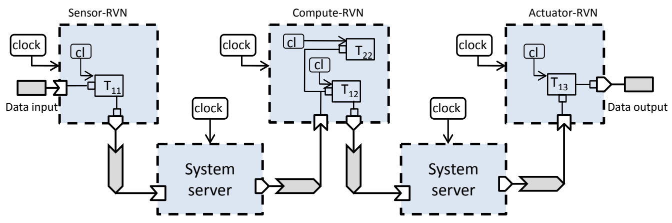


Figure 8.8: A Chain for server-based communication.

8.7.2 Computing the Worst Case Response Times of tasks

A prerequisite to compute the end-to-end latency is that the WCRT of each task in all RVNs. The computation of WCRT of tasks in RVN-servers is based on the periodic resource model presented in [15]. The overheads of server executions on FreeRTOS operating system have been calculated and included in the analysis presented in [18].

For a schedulable system, the WCRT for each task is calculated. The WCRT for a task is the time t at which `rbf` and `sbf` functions intersect at first and is calculated as follows:

$$\forall \tau_i,\ \texttt{smallest}\ t : 0 < t \leq D_i,\ \ \texttt{rbf}(i,t) = \texttt{sbf}(t), \tag{8.1}$$

where `sbf` is the supply bound function, $\texttt{rbf}(i,t)$ denotes the request bound function. `sbf` computes the minimum possible CPU supply to a server S_s for every time interval length t, and $\texttt{rbf}(i,t)$ of a task τ_i 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:

$$\texttt{rbf}(i,t) = C_i + b_i + \sum_{\tau_k \in \texttt{HP}(i)} \left\lceil \frac{t}{T_k} \right\rceil \cdot C_k, \tag{8.2}$$

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 `sbf` depends on the type of the overrun mechanism. In this paper we present results using overrun without payback using the following equations:

$$\texttt{sbf}(t) = \begin{cases} t - (k+1)(P_s - Q_s) & \text{if } t \in W^{(k)} \\ (k-1)Q_s & \text{otherwise,} \end{cases} \tag{8.3}$$

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 $[(k+1)P_s - 2Q_s, (k+1)P_s - Q_s]$.

The WCRT for all tasks presented in the Example in Section 8.7.1 are calculated using these formulas and presented in the last row of Table 8.2. All these formulas are implemented in the EELAP tool [7].

8.7.3 Computing data path analysis or end-to-end latencies

The end-to-end latencies for the communication chain $\tau_{11} \rightarrow \tau_{12} \rightarrow \tau_{13}$ for direct communication from Section 8.7.1 are computed and shown in the Figure 8.9. Since all RVNs are implemented on the same physical node, their clocks/periods are synchronized. Hence the activation times of the tasks are the same as of their periods. In Figure 8.9, the x-axis shows time line, the upward blue arrow and downward red arrow represent the task's activation and task's WCRT respectively, and the filled gray rectangle shows the time interval during which the task will be executed. The chain $\tau_{11} \rightarrow \tau_{12} \rightarrow \tau_{13}$ produces multiple outputs corresponding to a single input. The input of τ_{11} at time 40 (see Figure 8.9) produces four outputs at times 178, 198, 218, and 238; the first value at time 178 is new and presented by a bold arrow, while the following three are duplicates of this value and shown as normal arrows. The dotted arrow shows an overwritten value and is ignored.

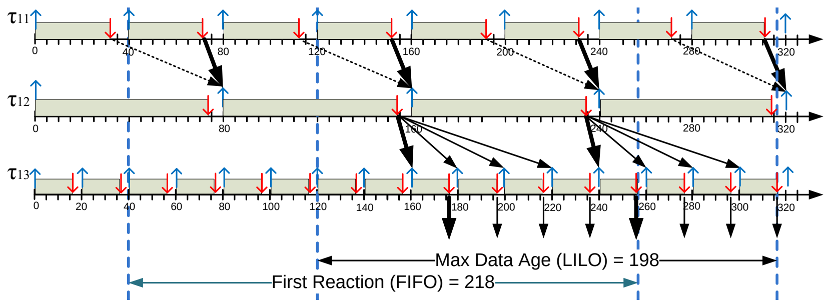


Figure 8.9: End-to-End latencies for the communication chain $\tau_{11} \rightarrow \tau_{12} \rightarrow \tau_{13}$.

Four different end-to-end latency semantics are identified in [26]. The most interesting ones are *first-reaction* and *max-data-age* (shown in Figure 8.9) which are highly used in embedded systems; like first-reaction is used in body electronics to find out how fast the response is, and max-data-age in control engineering to calculate maximum delays. *First-reaction* (or *first in first out (FIFO)*) is the time between the previous non-overwritten release of input task (τ_{11} at time 40) and the first output of last task in the chain corresponding to current non-overwritten release of input task (τ_{13} at time 258). It is the longest allowed time to produce the new data. *Max-data-age* (or *last in last out (LILO)*) is the time between the current non-overwritten release of input task (τ_{11} at 120) and its corresponding last output of last task in the chain (τ_{13} at 318). It is the longest time during which the data is allowed to age and the new data is produced after this time-limit.

The formulas and their corresponding algorithms to compute end-to-end semantics are explained in a technical report [6].

8.7.4 End-to-end latency analyzer for ProCom

We develop an analysis tool *End-to-End Latency Analyzer for ProCom (EE-LAP)* [7] to automate the computations of WCRTs of tasks and different end-to-end latency semantics for multi-rate server-based ProCom components. The tool performs the local and the global schedulability tests, and for a schedulable system it computes end-to-end latencies using the following two steps:

1. First it calculates the WCRTs of all tasks executing in a two-level hierarchical scheduling framework by using methods/formulas provided in Section 8.7.2,

2. And then it calculates different end-to-end latency semantics for the given communication chains for both communication strategies using algorithms/formulas provided in [33].

A technical report on the EELAP tool [33] presents the descriptions of API's of the tool, the formulas and their implementations in those API, and a user guide. It provides algorithms used to compute local and global schedulability condition, WCRTs for tasks, possible paths, path reachability, and different end-to-end latency semantics.

8.8 Case Study: Cruise controller and adaptive cruise controller

The PRIDE tool [12] supports the development of systems using ProCom components and we have used it for development of a cruise controller (CC) and an adaptive cruise controller (ACC) for automotive applications. Our motivating case study is simple, but exercises the execution-time properties and evaluates the integration and reusability of the run-time components.

The case study runs in two phases. First, the CC system is realized and exercised to test the temporal isolation 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 constant distance to 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 RVNs from the CC system. In the remainder of this section we describe the development of both applications.

8.8.1 System design

The CC system is designed using two ProSys components: Cruise Controller and Vehicle Controller. These ProSys components are deployed on two different virtual nodes `Virtual Node CC` and `Virtual Node VC` respectively, as shown in Figure 8.10. To extend the functionality to the ACC system, the Cruise Controller component is replaced with the Adaptive Cruise Controller component and is mapped to the `Virtual Node ACC`, while the `Virtual Node VC` is reused from the CC system (see Figure 8.10). These virtual nodes communicate with each other through input and output message ports.

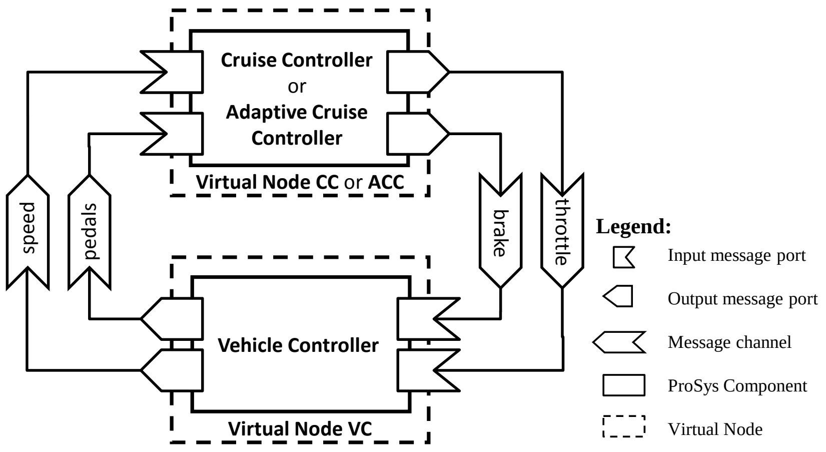
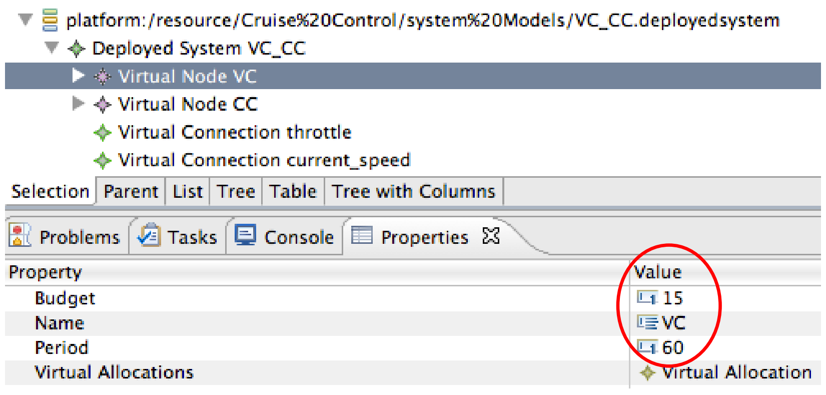


Figure 8.10: Deploying ProSys components on virtual nodes

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. The periods and budgets for these virtual nodes are assigned at the modeling level. The assignment of these values in the PRIDE tool is shown in Figure 8.11 and highlighted by circle. The detailed design of the ProSys components mentioned above is in turn shown in Figures 8.12, 8.13, and 8.14.

The Cruise Controller ProSys component contains three elements as shown in Figure 8.12: 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 ports accordingly, and an HMI Output to communicate the status to


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 8.11: The timing properties of the virtual node

the driver via the display. The Vehicle Controller ProSys component contains seven elements as shown in Figure 8.13: 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 the current speed to the output port periodically.

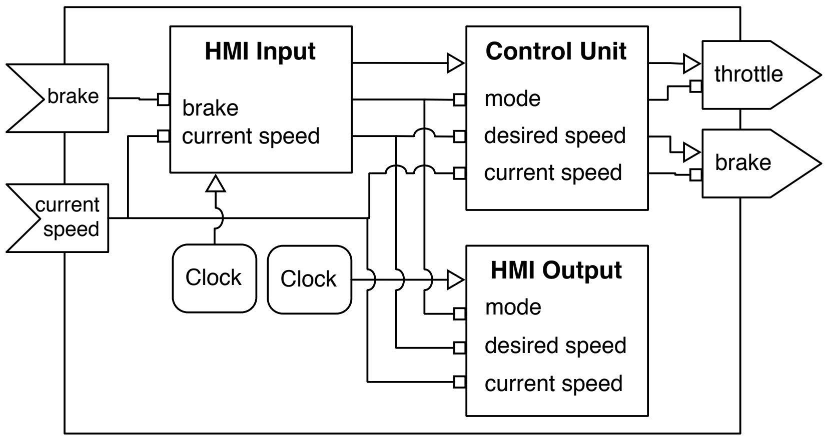


Figure 8.12: The Cruise Controller component

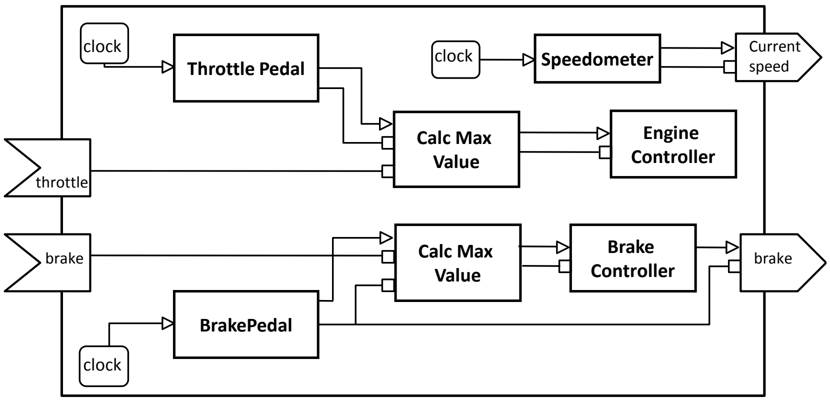


Figure 8.13: The Vehicle Controller component

The Adaptive Cruise Controller ProSys 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 8.14.

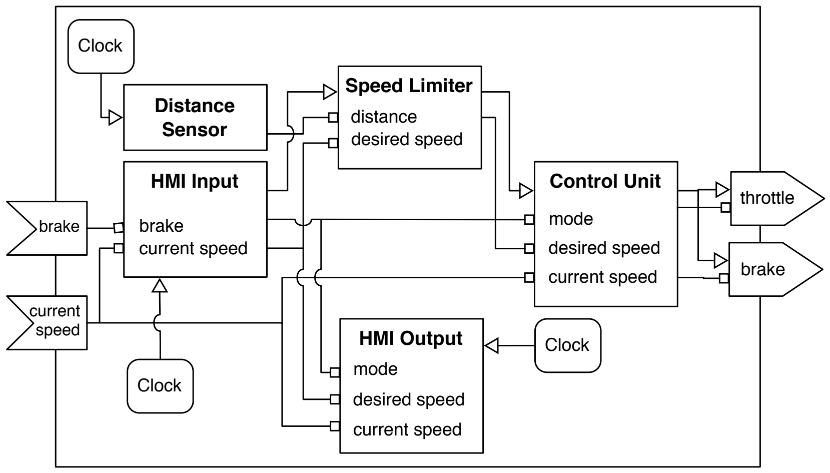


Figure 8.14: The Adaptive Cruise Controller component

8.8.2 Synthesis

As described in Section 8.5, the PRIDE tool automatically synthesizes code from the ProCom models at different stages. It takes the models as input, and generates all low-level platform independent code.

In the first step of the final synthesis/deployment process for the case study, two RVNs are produced for both CC and ACC systems: one RVN for `Virtual Node CC` or `Virtual Node ACC` and one for `Virtual Node VC`. These generated nodes contain tasks definitions.

One task is synthesized for each clock in the ProSys components. For the Cruise Controller component: `CCT1` task including HMI Input and Control Unit; and `CCT2` task including HMI Output component. Three tasks are generated for the Vehicle Controller 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. Three tasks are generated for the Adaptive Cruise Controller 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 RVNs (also called servers now) and to the tasks in them. Four servers are generated for both systems.

A `System` server is generated to provide communication among the RVNs. 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). The `System` server contains two tasks: a `Sender` and a `Receiver` task; whose functionality is to send and receive the data shared among RVNs 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.

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

Table 8.3: Servers used to test the CC and ACC systems behaviors.

The CC system contains two more servers in addition to `System` and `Idle` server: a `CC` server and a `VC` server associated with `Virtual Node`

CC and Virtual Node VC respectively. The ACC system also contains four servers: an ACC server associated with its Virtual Node ACC, it reuses the VC, System, and Idle servers from the CC system. The priorities, periods and budgets for these servers are given in Table 8.3.

Our implementation supports both idling periodic and deferrable servers, however, in this paper we are showing results with only idling periodic server. 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 the 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 is ready to execute. Task properties and their assignments to the servers are given in Table 8.4.

Tasks	CCT1	CCT2	ACCT1	ACCT2	ACCT3	VCT1	VCT2	VCT3	Sender	Receiver
Server	CC	CC	ACC	ACC	ACC	VC	VC	VC	$SYSTEM$	$SYSTEM$
Priority	2	1	2	2	1	1	1	2	2	2
Period	40	60	40	40	60	60	60	40	20	20

Table 8.4: Tasks properties and their assignment to servers.

Once all the platform dependent user code is finalized, all RVNs 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 a physical node. Currently the PRIDE tool is evolving and the automatic synthesis part is not fully mature. Hence a few parts of these experiments were synthesized manually, but it is not relevant for our experiments and does not effect our results.

8.9 Evaluation and discussion

In this section we evaluate the timing properties of RVNs and the communication strategies.

8.9.1 Evaluating timing properties of RVN

For the evaluation in this section we use the servers and task-sets synthesized in our case study in Section 8.8, and are summarized in Table 8.3 and Table 8.4.

We have performed the experimental evaluation on an AVR-based 32-bit EVK1100 board [8]. 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 FPPS scheduling policy at both levels for idling periodic servers. Its tick-handler runs at the rate of 1ms.

Our evaluation focuses mainly on 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 final binaries for both systems are executed on the micro-controller and the traces of executions are visualized. The experimental results are presented in the form of visualization of servers executions in Figures 8.15, 8.16, 8.17 and 8.18.

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 vertical axis shows the server's remaining 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 execute until their budgets are depleted, if no task is ready then the idle task of that server executes till its budget is depleted[1].

Testing temporal isolation and predictable integration

To test the temporal isolation among RVNs and their predictable integrations, the CC system is synthesized with the previously described four servers and task sets belonging to those servers. The servers executions (according to their resource reservations) along with their task sets are presented in Figure 8.15 and Figure 8.16 [10].

Figure 8.15 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 RVNs. For example, if one server (RVN) is overloaded and its tasks miss deadlines, it should not affect the behavior of other servers(RVN) in the system.

The same example is executed to perform this test but with the increased utilization of the CC server as shown in Figure 8.16. The execution times of tasks CCT1 and CCT2 are increased by adding busy loops, hence making the

[1]Due to space reasons, we have not visualized the results for deferrable servers here. They are presented in [5].

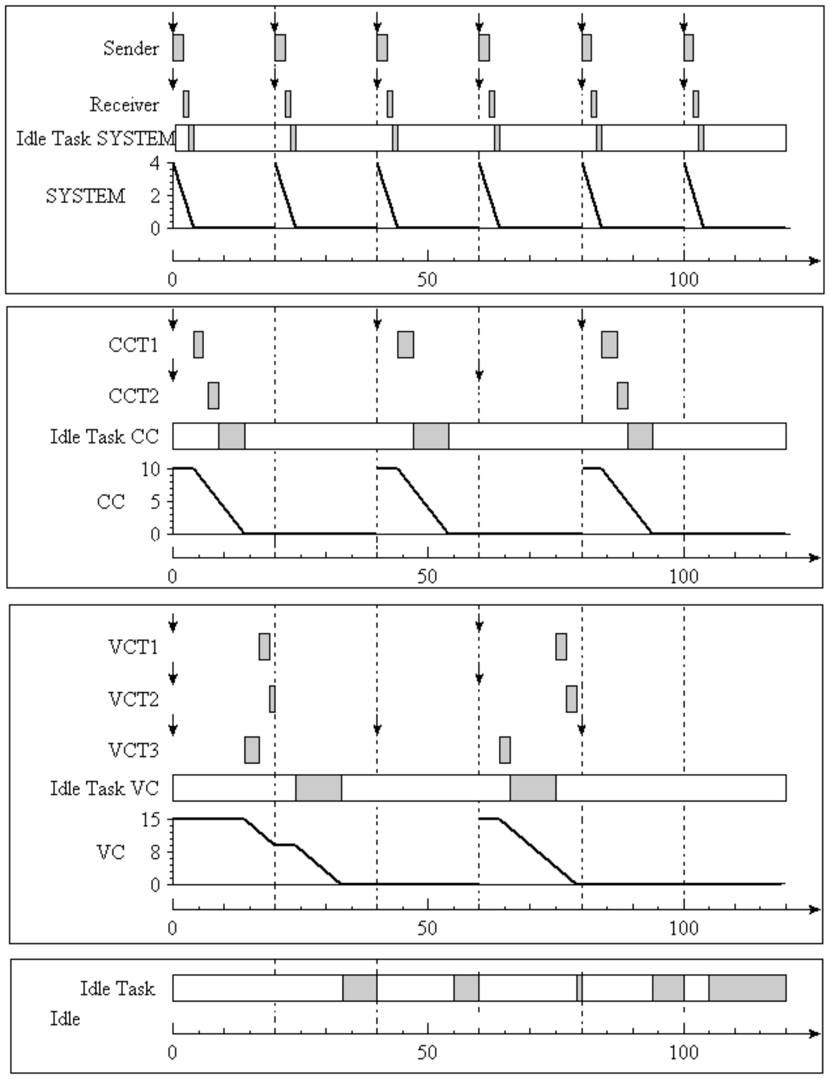


Figure 8.15: The trace for servers in the CC system during normal load

`CC` server's utilization greater than `1`. Therefore the low priority task `CCT2`

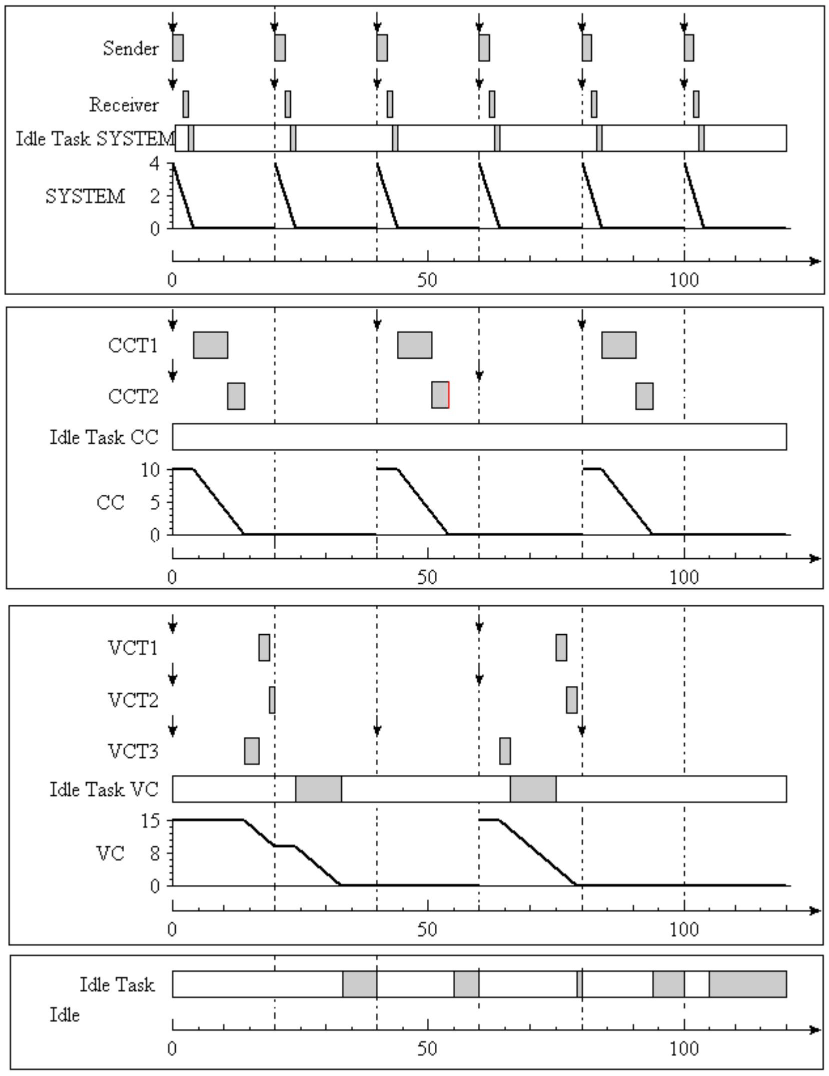


Figure 8.16: Trace showing temporal and fault isolations during overload situation

misses its deadlines. 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. But at time 54 CCT2 is preempted again due to it's server's budget expiration and will miss its deadline. Further, the CC is never idling because it is overloaded (Idle task of CC server is not executed in Figure 8.16).

The overload in the CC server does not effect the behavior of any other server in the system as obvious from Figure 8.16. 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, RVNs exhibit a predictable timing behaviour that eases their integration. It also manifests that the temporal errors are contained within the faulty RVN only and their effects are not propagated to the other RVNs in the system.

Testing component's reusability

The purpose of this experiment is to test the reusability of RVNs 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 8.17.

Since the RVNs 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 8.17 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 changed timing properties, i.e. period 50 and budget 14 as shown in Figure 8.18. As long as the allocated budgets to servers (at the modeling level) are provided, the timing properties are guaranteed at the execution. In Figure 8.18 we observe a change in the response times of tasks of the low priority VC server due to the changed timing behaviour of the high priority ACC server, but still all tasks of VC server meet their deadlines.

Hence, by using RVN components and two-level deployment process, the timing requirements are also encapsulated within the components along with their functional requirements and the temporal partitioning is provided among

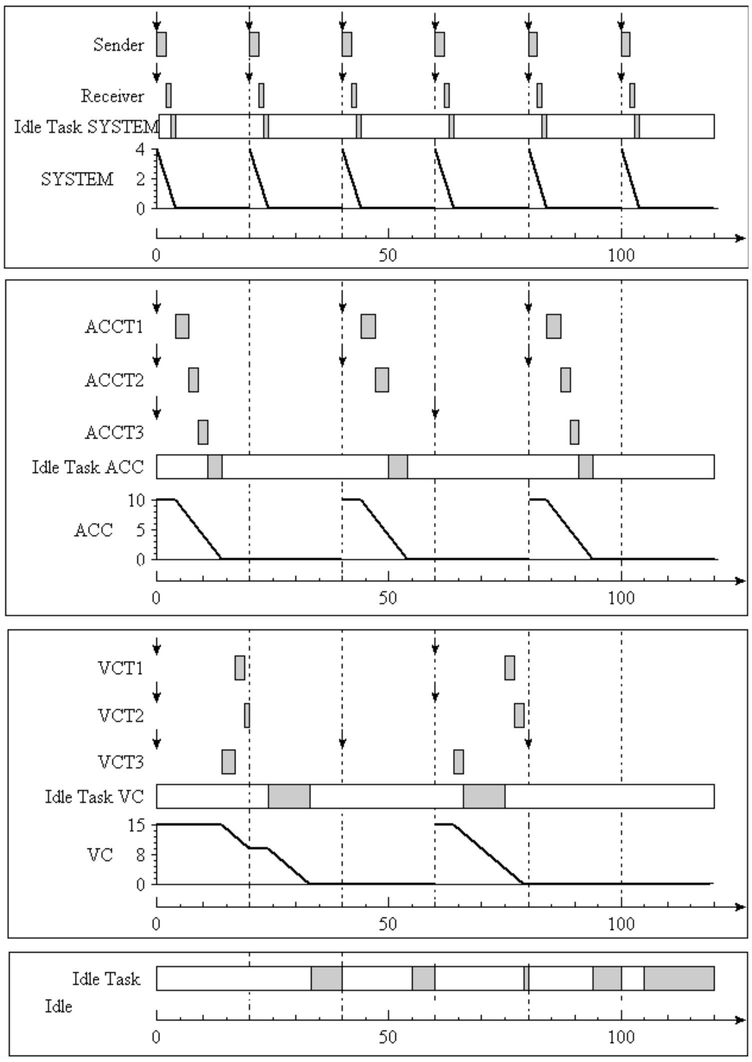


Figure 8.17: Trace showing reusability of runnable virtual nodes in ACC system

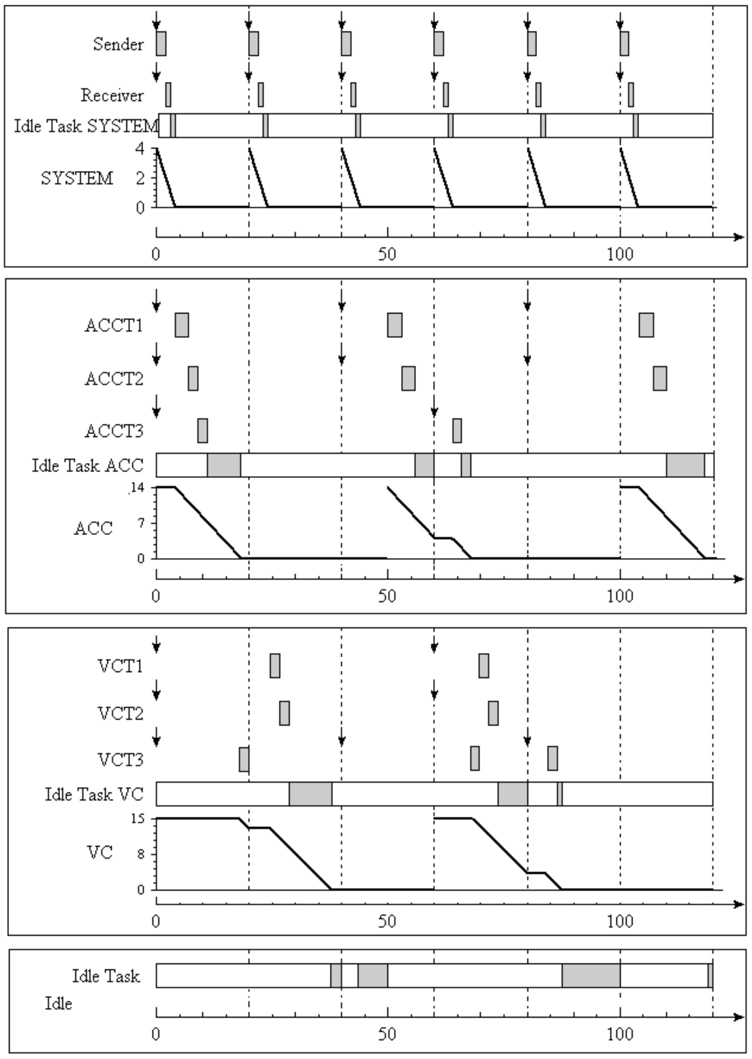


Figure 8.18: Trace showing predictable integration and reusability of RVNs in ACC system

the components (using HSF), that results in the increased predictability during component's integration and making the runnable virtual nodes reusable entities.

8.9.2 Evaluating communication strategies

For the evaluation in this section we use the EELAP tool to compute the end-to-end latencies for both direct and server-based communication strategies. There is a trade-off between both strategies: the direct strategy is fast but is more restricted in the reuse of real-time components; on the other hand, server-based strategy provides reusability, maintainability, and flexibility to change the communication patterns but suffers from longer end-to-end delays. In our evaluations, we mainly focus on 1) evaluating the increase in end-to-end latencies for server-based strategy and 2) revealing different factors that affect these latencies for server-based strategy, so that by restraining these factors the latencies could be minimized.

Comparing end-to-end latencies for server-based and direct communications

We calculate different end-to-end latencies for the Example given in Section 8.7.1 and summarized in Table 8.1 and Table 8.2 for both direct (Figure 8.7) and server-based (Figure 8.8) communication chains. The System server attributes used for the calculation are `period` 12, `budget` 3, `and priority` 7, and its tasks attributes are given in Table 8.5.

`Tasks`	`sender`	`receiver`
`Period`	20	20
`Exe. Time`	1	1
`Priority`	2	1
`WCRT`	19	20

Table 8.5: Timing properties of tasks in System servers.

The timing attributes of all servers and their tasks, and the communication chain are input to the tool. For both communication strategies, the values for first-reaction and data-age latencies are computed for a small chain $\tau_{11} \rightarrow \tau_{13}$ and a long chain $\tau_{11} \rightarrow \tau_{12} \rightarrow \tau_{13}$ and are provided in the Table 8.6. For server-based strategy, the inclusion of an extra System server in the communication results in longer chains: small chain becomes $\tau_{11} \rightarrow$ `sender` $\rightarrow$ `receiver`

$\rightarrow \tau_{13}$, and long chain becomes τ_{11} $\rightarrow$ `sender` $\rightarrow$ `receiver` $\rightarrow \tau_{12} \rightarrow$ `sender` $\rightarrow$ `receiver` $\rightarrow \tau_{13}$ therefore, the latencies for server-based strategy are longer than that of direct strategy. In Table 8.6, the column `Value` provides the first-reaction and data-age latencies' values while the column `Increase` represents the percentage-increase in the end-to-end latencies for server-based strategy as compared to that of direct strategy for both small and long chains. For this particular example the increase is less than 30%.

Chain	Strategy	First-reaction		Data-age	
		Value	Increase	Value	Increase
Small chain	server-based	138	28.9%	108	27.8%
	direct	98		78	
Long chain	server-based	278	21.6%	258	23.2%
	direct	218		198	

Table 8.6: Comparison between latencies for both communication strategies.

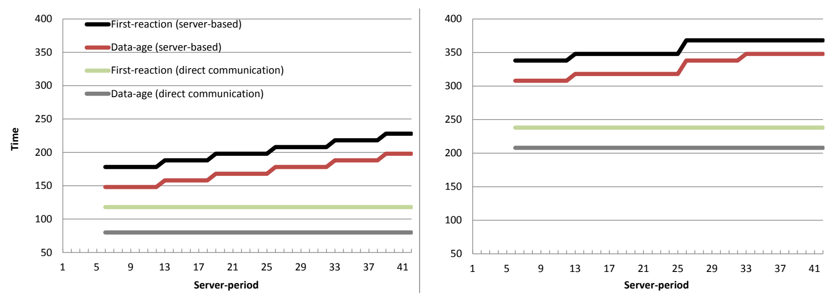


Figure 8.19: Results of End-to-End latencies for different System server-periods for small (left) and long (right) communication chains.

Revealing factors influencing end-to-End latencies in the server-based communication

The first obvious factor causing an increase in end-to-end latencies is the inclusion of an extra server. For longer chains we need to include the System server in the communication chain more than once, for example $\tau_1 \rightarrow \tau_2$ chain needs one time inclusion, while $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ chain needs two times inclusion of the

System server. This factor cannot be restrained since the communication chain cannot be reduced.

The second factor increasing the latencies is the *period of System server and its tasks*. Longer period means that the server is activated after long time hence increasing end-to-end communication latencies. Moreover, for longer latencies, the data becomes very old which could cause invalidity of data. Thus it is important to restrain this factor. This can be done by choosing suitable periods for System server and its tasks.

To reveal suitable ranges of periods for System server and its tasks for the server-based strategy, we have performed experiments using the EELAP tool. The server-period length depends on its tasks periods and is usually half than the task period [15]. Therefore, in these experiments, periods for sender and receiver tasks are kept fixed and System server-period is kept variable for a certain period range (the range at which the system is schedulable). The end-to-end latencies are computed for the Example given in Section 8.7.1. The `sender` and `receiver` tasks periods are set long i.e. 80 for both while their execution times and priorities are the same as given in Table 8.5. The experiments are done on a varying server-period range (i.e. from 6 to 42) as the system is schedulable within this range only. The first-reaction and max-data-age latencies are computed for a small chain ($\tau_{11} \rightarrow$ `sender` $\rightarrow$ `receiver` $\rightarrow \tau_{13}$ for server-based, $\tau_{11} \rightarrow \tau_{13}$ for direct) and a long chain ($\tau_{11} \rightarrow$ `sender` $\rightarrow$`rec eiver` $\rightarrow \tau_{12} \rightarrow$ `sender` $\rightarrow$ `receiver` $\rightarrow \tau_{13}$ for server-based, and $\tau_{11} \rightarrow \tau_{12}$ $\rightarrow \tau_{13}$ for direct strategy).

The results for small and long chains are plotted as line graphs in Figure 8.19 on left and right sides respectively. In this figure, the server-period increases from left to right along the horizontal axis, and the time is represented along the vertical axis. The graph shows that latencies for long chains are increased as compared to the latencies of small chain for both communication strategies. The increase of latencies for the direct strategy is slightly less than the increase for server-based strategy. The graph also reveals the effects of longer System server-period on the latencies for the server-based strategy. It is clear that (1) the increase in End-to-end latencies is linear; (2) the increase in latencies is much higher when `sender` and `receiver)` tasks' periods are longer than the other tasks in the system. Hence it is better to keep the periods of System server and its tasks as short as possible.

The server-based strategy provides the benefits of reusability, maintainability and extendibility of real-time components with a slight overhead of communication, and is a better strategy to use in systems where the reusability requirement is high. These results manifest that the overhead of communication for the server-based strategy could be minimized by selecting shortest possible periods for the System server and its tasks.

8.10 Related work

We describe some contemporary component-technologies available for embedded systems focusing on the deployment, integration and reuse of components and on the predictability in time-domain of the resulting systems.

8.10.1 Temporal isolation and predictable integration among components

ARINC-653 is used as a platform to implement partitioning for avionics software with emphasis on predictability and safety-critical issues [27]. It provides fully deterministic top-level Time Division Multiple Access (TDMA) based schedule. Some work has done to build a platform that combines component-based software construction with hard real-time operating system services to manage the challenge of increasing complexity of hard real-time systems by employing reusable components and robust composition techniques. CORBA Component Model (CCM) [28] is extended and combined with ARINC-653 to build a platform using Linux processes and POSIX threads [29]. Integrated Modular Avionics (IMA) architecture also encourages the integration of software functions and their reuse [30]. Our RVN concept can also be deployed in one of the ARINC partitions.

All these models provide spatial and temporal partitioning between applications for fault containment, using a two-tier model like RVN. However, in ARINC, the top-level uses only fixed temporal scheduling of software partitions (TDMA slots are fixed for repetitive execution of major frames) whereas in RVNs it is flexible; i.e. the RVN-server parameters can be changed as long as the task set is schedulable. The reuse of ARINC partitions is also restricted due to the fixed time slots, the partitions could be moved to other platforms only if the TDMA slot matches exactly. RVNs have more freedom of reuse because RVNs can be used with the changed timing properties (as long as the task set is schedulable).

Architecture Analysis and Design Language (AADL) was developed as an SAE Standard AS-5506 [31]. The Ocarina [32] tool suite facilitates the design of AADL component 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, (e.g. PolyORB or PolyORB-HI for code generation in Ada or C respectively) [33]. A process contains many tasks and is a self-contained runnable entity executable on a hardware platform. Another type of runnable entity for AADL employs a hierarchical scheduling concept in a partition, provides temporal isolation among runnable entities and is supported by a tool suite POK [34], which is ARINC compliant. Unlike ProCom, deployment in AADL is done in a single step to directly execute the generated code on hardware, without any consideration of executable component's reusability.

Ptolemy architecture [35] provides temporal correct composition of model elements from its associated modeling language by global system-synthesis. For temporal predictability the whole system is synthesized together and executed by a scheduler, which manages all the events expressing progression of the Ptolemy-model. It lacks composition of run-time artefacts and requires a special purpose execution engine to achieve temporal correctness. Inclusion of legacy components which are not developed using Ptolomy-language is difficult and produces unpredictable behavior.

Time-Triggered Architecture [36] provides a concept where each component is statically scheduled. Thus, integration of components causes minimal modification to the timing behavior of each component. However, the static nature limits the usability of the approach and poses large difficulties to achieve high utilization of hardware resources. Also, integration of components with conflicting schedules requires rescheduling.

8.10.2 Temporal reuse of components

AUTomotive Open System ARchitecture (AUTOSAR) [37] is an open standard for automotive electronics architectures with the principal aim of the standard is to master the growing complexity of automotive electronic and software architectures. Its main disadvantage is the lack of clear and well-defined timing properties that further affect the execution semantics and real-time components' integration. It requires additional testing of timing properties when components are reused. On the other hand, ProCom puts special focus on such requirements right from the beginning of the component's development until the component's deployment. TIMMO (TIMing MOdel) project provide timing model for AUTOSAR and is included in the version 4.0 of AUTOSAR specification [38]. TADL (Timing Augmented Description Language) [39] is used to express the timing requirements and timing constraints in all design phases during the development of embedded software. TADL is extended to

TADL2 in TIMMO-2-USE project [40], which not only refines the previous TADL and updates the timing constraints but also provides new algorithms, tools and a methodology to model advanced timing at different levels of abstraction in compliance with the AUTOSAR-based tool chain [38]. 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, leveraging HSF to preserve internal temporal behavior during integration and reuse.

The Rubus Component Model (RCM) [41] is similar to ProCom in many aspects: like capturing the functionality at two-levels of component hierarchy; managing different ports for control and data flows; graphical design tools, a scheduler, and some plug-ins to perform analysis. Its main difference from the ProCom technology is at the deployment and execution levels. In Rubus, the required hardware components are explicitly modelled, and therefore, are highly restricted in reuse, some metadata facilitate temporal reuse but do not explicitly focus on it unlike the ProCom where the components are developed independently from the hardware details and hardware specifics are taken care at the last step of deployment facilitating temporal reuse of executable components. Further, there is no notion of temporal/hierarchical partitioning in Rubus. Thus, Rubus provides composition of model-elements but not of runtime artefacts.

The Deployment and Configuration (D&C) of component-based distributed applications [42] is standardized by the OMG to facilitate the deployment of component-based applications onto target platform. It uses a Platform Independent Model (PIM) for the model components, and a Platform Specific Model (PSM) for the CORBA Component Model (CCM) [28]. The Real-time D&C (RT-D&C) [43] is its extension to develop applications with real-time properties and provide a deployment plan. The metadata about the temporal behavior of components is added to the specification at the Platform Independent Model (PIM) level to facilitate the real-time analysis of the components. However, a RT-planner configures the timing properties of real-time application after using the real-time analysis tools, which is different from the ProCom deployment where the timing properties are preserved within the executable RVNs. Unlike RVN, in RT-D&C components, the timing properties are preserved and reusable till the component's development at PIM level not until the execution level.

8.11 Conclusions

This paper presents the executable component runnable virtual nodes (RVNs) as a means to achieve predictable integration and reuse of software components using a two-level deployment process for real-time software components. The RVN is intended as a coarse-grained component for a single node deployment with internal multitasking. The notion of two-level deployment process encapsulates the timing properties and uses the hierarchical scheduling framework (HSF) within RVNs to preserve timing behaviour. The HSF provides temporal separation and the ProCom component-model provides functions separation between RVNs. Thus, our approach allows unit-testing and -analysis with respect to both temporal- and functional-properties. These properties are then preserved when the RVN is integrated with other components. Compared to previous work [10], results have been extended by allowing these RVN properties to be reused in a new environment, thereby facilitating the development of complex real-time systems.

In addition to [10], a server-based communication strategy is implemented that supports predictable integration and reuse of the timing properties of RVNs by keeping communication-code in a separate server. This strategy incorporates the maintainability and flexibility to change the communication code without affecting the timing properties of RVNs. We have evaluated the end-to-end delay analysis for the server-based strategy with a more direct communication strategy for efficiency and reusability properties of RVNs. Hence using RVNs and server-based inter-RVN communication, complex real-time systems can be developed as a set of well defined reusable components encapsulating functional and timing properties.

Finally, an extended proof-of-concept case study demonstrates the temporal-fault containment within an RVN as well as the reuse of RVNs in new environment. The work is based on the ProCom component-technology running on the HSF implementation on FreeRTOS and is executed on an AVR-based EVK1100 board. However, we believe that our concept is applicable also to commercial component technologies like AADL, AUTOSAR [9].

For future work, we plan to automate the code-generation for communication between physical nodes and support run-time migration of RVNs between physical nodes. We also plan to support virtual communication-channels using server-based scheduling techniques for e.g. CAN [44]. This will allow development, integration and reuse of distributed components using a set of RVNs and virtual buses.

Bibliography

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

[2] 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.

[3] T. Nolte. Compositionality and CPS from a platform perspective. In *Proceedings of the 1st International Workshop on Cyber-Physical Systems, Networks, and Applications (CPSNA'11)*, August 2011.

[4] S. Sentilles, A. Vulgarakis, T. Bureš, J. Carlson, and I. Crnković. A component model for control-intensive distributed embedded systems. In *11th International Symposium on Component Based Software Engineering*, pages 310–317, 2008.

[5] R. Inam, J. Mäki-Turja, M. Sjödin, S. M. H. Ashjaei, and S. Afshar. Support for hierarchical scheduling in FreeRTOS. In *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA' 11)*, Tolouse, France, September 2011. IEEE Computer Society.

[6] Jiří Kunčar, Rafia Inam, and Mikael Sjödin. End-to-End Latency Analyzer for ProCom - EELAP. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-272/2013-1-SE, Mälardalen University, School of Innovation, Design and Engineering, 2013.

[7] Jiří Kunčar. End-to-End Latency Analyzer for ProCom - EELAP, March 2013. https://github.com/jirikuncar/eelap/, last checked: 20.03.2013.

[8] ATMEL EVK1100 product page. Website, 2013. http://www.atmel.com/tools/evk1100.aspx, last checked: 20.03.2013.

[9] R. Inam, J. Mäki-Turja, J. Carlson, and M. Sjödin. Virtual Node – to achieve temporal isolation and predictable integration of real-time components. *International Journal on Computing (JoC)*, 1(4), 2012.

[10] R. Inam, J. Mäki-Turja, M. Sjödin, and J. Kunčar. Real-time component integration using Runnable Virtual Nodes. In *38th Euromicro Conference on Software Engineering and Advanced Applications (SEAA' 12)*, Izmir, Turkey, September 2012. IEEE Computer Society.

[11] Rafia Inam. *Towards a Predictable Component-Based Run-Time System*. Number 145. Licentiate thesis, January 2012.

[12] PRIDE Team. PRIDE: the PROGRESS Integrated Development Environment, 2010. "http://www.idt.mdh.se/pride/?id=documentation, last checked: 20.03.2013".

[13] E. Borde and J. Carlson. Towards verified synthesis of ProCom, a component model for real-time embedded systems. In *14th ACM SIGSOFT Symposium on Component Based Software Engineering*, 2011.

[14] J. Carlson, J. Feljan, J. Mäki-Turja, and M. 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)*, 2010.

[15] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proc. 24th IEEE Real-Time Systems Symposium (RTSS'03)*, pages 2–13, 2003.

[16] Richard Barry. *Using the FreeRTOS Real Time Kernel*. Real Time Engineers Ltd., 2010.

[17] FreeRTOS web page. Web-site, 2013. http://www.freertos.org/, last checked: 20.03.2013.

[18] R. Inam, J. Mäki-Turja, M. Sjödin, and M. 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.

[19] L. Sha, J.P. Lehoczky, and R. Rajkumar. Solutions for some practical problems in prioritised preemptive scheduling. In *Proc. 7th IEEE Real-Time Systems Symposium (RTSS'86)*, pages 181–191, 1986.

[20] 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.

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

[22] R. I. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *Proc. 27^{th} IEEE Real-Time Systems Symposium (RTSS)*, pages 389–398, 2006.

[23] M. Behnam, T. Nolte, M. Sjödin, and I. Shin. Overrun methods and resource holding times for hierarchical scheduling of semi-independent real-time systems. *IEEE Transactions on Industrial Informatics*, 6(1), 2010.

[24] R. Inam and M. Sjödin. Implementing and evaluating communication-strategies in the ProCom component technology. In *24th Euromicro Conference on Real-Time Systems (ECRTS 2012), WiP*. ACM SIGBED Review, July 2012.

[25] M.D. Natale, W. Zheng, P. Giusto, and A. S. Vincentelli. Optimizing End-to-End latencies by adaption of the activation events in distributed automotive systems. In *13th IEEE Real Time and Embedded Technology and Applications symposium (RTAS'07)*, 2007.

[26] N. Feiertag, K. Richter, J. Nordlander, and J. Jonsson. A compositional framework for End-to-End path delay calculation of automotive systems under different path semantics. In *Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS'08)*, November 2008.

[27] S. Han and H.W. Jin. Full virtualization based ARINC - 653 partitioning. In *30th IEEE/AIAA Digital Avionics Systems Conference*, 2011.

[28] Object Management Group. CORBA component model specification, OMG available specification, Version 4.0, April 2006. http://www.omg.org/spec/CCM/4.0/PDF/.

[29] A. Dubey, G. Karsai, R. Kereskenyi, and N. Mahadevan. A real-time component framework: experience with CCM and ARINC-653. In *13th IEEE ISORC*, 2010.

[30] L. M. Kinnan. Application migration from LINUX prototype to deployable IMA platform using ARINC-653 and OPEN GL. In *IEEE/AIAA 26th Digital Avionics Systems Conference (DASC '07)*, 2007.

[31] SAE International. AADL specification, 2006. http://standards.sae.org/as5506/1/, last checked: 20.03.2013.

[32] 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.

[33] Thomas Vergnaud, Jrme Hugues, Laurent Pautet, and Fabrice Kordon. PolyORB: a schizophrenic middleware to build versatile reliable distributed applications. In *Reliable Software Technologies - Ada-Europe 2004*, Lecture Notes in Computer Science, pages 106–119. 2004.

[34] J. 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.

[35] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S.Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity-the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[36] H. Kopetz and G. Bauer. The Time-Triggered Architecture. *Proceedings of the IEEE*, 91(1):112–126, January 2003.

[37] AUTOSAR GbR. Specification of operating system, 2008. http://www.autosar.org/download/, last checked: 20.03.2013.

[38] TIMMO project. Mastering timing information for advanced automotive systems engineering, in the TIMMO-2-USE brochure, September 2012. http://www.timmo-2-use.org/pdf/T2UBrochure.pdf, last checked: 20.03.2013.

[39] TADL. TADL: Timing Augmented Description Language, Version 2. Deliverable 6, October 2009. The TIMMO Consortium.

[40] TIMMO project. TIMMO-2-USE, 2013. http://www.timmo-2-use.org/, last checked: 20.03.2013.

[41] K. Hänninen, J. M-Turja, M. Nolin, M. Lindberg, J. Lundbäck, and K. L. Lundbäck. The Rubus component model for resource constrained real-time systems. In *3rd International Symposium on Industrial Embedded Systems*, 2008.

[42] Object Management Group. Deployment and Configuration of component-based distributed applications specification, 2006. v4.0.

[43] P. L. Martinez, C. Cuevas, and J. 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.

[44] T. Nolte, M. Nolin, and H. Hansson. Real-time server-based communication for CAN. *IEEE TIE*, 1(3):192–201, April 2005.

Chapter 9

Paper D: The Multi-Resource Server for predictable execution on multi-core platforms

Rafia Inam, Nesredin Mahmud, Moris Behnam, Thomas Nolte, Mikael Sjödin.
In the 20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'14), pages 1-11, IEEE, Berlin, Germany, April, 2014.

Abstract

In this paper we present an implementation and demonstration of the Multi-Resource Server (MRS) which enables predictable execution of real-time applications on multi-core platforms. The MRS provides temporal isolation both between tasks running on the same core, as well as, between tasks running on different cores. The latter could, without MRS, interfere with each other due to contention on a shared memory bus.

We demonstrate that MRS can be used to ”encapsulate” legacy systems and to give them enough resources to fulfill their purpose. In our case study a legacy media-player is integrated with several resource-hungry tasks running at a different core. We show that without MRS the media-player starts to drop frames due to the interference from other tasks; while introduction of MRS alleviates this problem. Another part of our demonstration shows how traditional periodic real-time tasks can be kept schedulable even when tasks with high memory-demand are added to the system.

9.1 Introduction

Using multi-cores for real-time applications presents many challenges. One such challenge is to achieve and maintain predictable execution of concurrent tasks that compete for both CPU- and memory-bandwidth resources. On uni-core platforms, the server-based scheduling approach successfully bounds the interference between the applications running [1, 2, 3]. However, this approach is limited to provisioning of the CPU resource only and it does not take the memory bandwidth problem into account, the latter problem often being inherited from when migrating software from a single-core to a multi-core architecture. In this paper we target statically partitioned multi-core real-time systems. For these systems we present the Multi-Resource Server (MRS) technology that schedules the two resources CPU- and memory-bandwidth, in order to achieve a predictable execution of embedded real-time systems.

In statically partitioned multi-core systems, concurrent tasks allocated to the same core interfere with each other by competing for CPU-bandwidth (we call this *local interference*), and concurrent tasks allocated to different cores interfere by competing for memory-bandwidth (we call this *global interference*). In addition to these sources of interference, tasks can also experience both local and global cache-pollution interference. If needed, e.g. for hard real-time systems, cache pollution can be relieved by cache-partitioning techniques like [4, 5], which are not within the scope of this paper. Thus, the implementation of the MRS presented here is suitable for soft real-time systems. However, if cache pollution can be avoided (e.g., by cache partitioning [5] or by disabling caches, or bounding caches by some static analysis technique [6, 7]), the schedulability analysis presented in [8] paves the way for using MRS also in hard real-time systems.

Additionally, we practically demonstrate the capability of MRS to maintain a predictable execution of a legacy soft real-time application. We show that MRS is not only useful when developing new systems; it is also useful to encapsulate and protect legacy applications, e.g., when performing a migration of applications from a single core to a multi-core platform. While our example uses a single task, the MRS allows for a complete subsystem with a set of tasks (and potentially its own scheduling algorithm) to be encapsulated in a *server* and then share an allocation of CPU- and memory-bandwidth resources. This is later demonstrated in a case-study using a synthetic setup.

We have presented the basic idea of the MRS approach in [8] where a theoretical analysis framework is provided to assess the composability of applications/subsystems. In this paper, we focus on the implementation of the server,

and its evaluation using (1) a case study and (2) a synthetic experimental setup. Partitioned scheduling is considered in which servers and tasks are statically allocated to a specific core. The rationale for looking at statically partitioned systems is due to our industrial partners' preference.
The main contributions of this paper are:

- We present the first implementation of the MRS[1]. This implementation is made as a user-space library for Linux running on COTS hardware.
- We demonstrate how the MRS can be used to preserve the functionality of a legacy application when it is executed on a single core while another core executes tasks with adverse memory behavior.
- We demonstrate for a synthetic task-set how the MRS can be used to isolate tasks from each other to prevent adverse behavior of some tasks to negatively impact other tasks.
- We measure the overhead of memory related parts of the MRS and we conclude that it is low.

Paper Outline: Section 9.2 explains system model, followed by Section 9.3 that describes the MRS concept in details. A brief overview of the software framework used to implement the MRS is presented in Section 9.4. Implementation details are covered in Section 9.5. Section 9.6 presents the evaluation setup and Section 9.7 describes a case study using a soft real-time application. Synthetic evaluations are performed and results are analyzed in Section 9.8. Section 9.9 presents the related work, and finally, Section 9.10 concludes the paper.

9.2 System model

In this section we present our target hardware platform, the system model that we use, and the assumptions that we follow.

9.2.1 Architecture

In our work we assume the architecture to consist of a processor with a set of identical cores that all have uniform access to the main memory. Each core has a set of local resources; primarily a set of caches for instructions and data. The

[1]The MRS implementation is available as an open source project at `http://www.idt.mdh.se/~MemSched/`

system has a set of resources that are shared amongst all cores; this will typically be the last-level cache, main-memory and the shared memory bus. Our architecture is industrially relevant and is the first step towards more advanced architectures.

We assume that a local cache miss is stalling, which means that whenever there is a miss in a local cache the core is stalling until the cache-line is fetched from memory. We focus on the shared memory bus and we assume that all accesses to the shared memory and the last-level cache go through the same bus, and that the bus serves one request at a time. It is worth noticing that any single-core could easily generate enough memory traffic to saturate the memory bus by executing memory intensive tasks.

9.2.2 Server model

Our scheduling model for the multi-core platform can be viewed as a set of trees, with one parent node and many leaf nodes per core, as illustrated in Figure 9.1. The parent node is a *node scheduler* and leaf nodes are the servers. Each server has its own set of tasks that are scheduled by a *local scheduler*. The node scheduler is responsible for dispatching the servers according to their bandwidth reservations (which include both CPU- and memory-bandwidth). The local scheduler then schedules its task set according to a server-internal scheduling policy.

Each server S_s is allocated a budget for CPU- and memory-bandwidth according to $\langle P_s, Q_s, M_s \rangle$, where P_s is the period of the server, Q_s is the amount of CPU-time allocated to the server each period, and M_s is the number of allowed memory requests in each period. The CPU-bandwidth of a server is thus Q_s/P_s and we assume that the total CPU-bandwidth for *each core* is not more than 100%. M_s/P_s is the memory-bandwidth for S_s, and we assume that the total memory-bandwidth allocated to servers *on all cores* is not more than what can be served on the shared memory bus. Server parameters can be obtained using analysis [8] or from domain expertise.

During run-time each server is associated with two dynamic attributes q_s and m_s which represent the amount of available CPU- and memory-budgets respectively. The implementation in this paper uses *Fixed Priority Pre-emptive Scheduling (FPPS)* policy for both node scheduling and server scheduling.

We assume that each server is assigned to one core and that its associated tasks will always execute only on that core i.e., we use the partitioned multiprocessor scheduling technique. The terms memory-bandwidth reservation and memory reservation are used interchangeably in the rest of the paper.

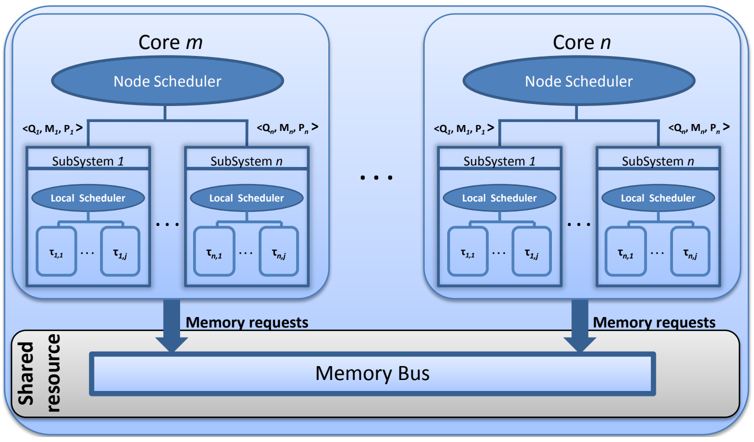


Figure 9.1: The multi-resource server model

9.2.3 Task model

We are considering a simple sporadic task model in which each task τ_i is represented as $\tau_i(T_i, C_i, D_i)$ where T_i denotes the minimum inter-arrival time of task τ_i with Worst-Case Execution Time $WCET_i$ and deadline D_i, where $D_i \leq T_i$. Each task τ_i has a fixed priority ρ_i. During the execution of tasks, memory requests can be made arbitrary at any time which can cause cache misses, i.e., the model of memory requests of each task is not known in advance.

9.3 The multi-resource server

The goal of the MRS is to provide temporal isolations through resource reservation approaches in the context of CPU bandwidth reservation [9] and memory bandwidth reservation [10]. The following subsections explain the MRS server and mechanisms used to manage memory budget of the server.

9.3.1 The MRS mechanism

We explain the MRS using the following rules:
Rule 1: The MRS server is of periodic type, i.e., it replenishes both CPU- and memory-budgets to the maximum values periodically. At the beginning of each

server period its dynamic attributes are set as $q_s = Q_s, m_s = M_s$.
Rule 2: In each core, a node scheduler is responsible to schedule all ready servers. A server is in the *ready state* if its remaining budgets are greater than zero, i.e. $q_s > 0$ and $m_s > 0$. The scheduled server applies its associated local scheduler to schedule its ready tasks.
Rule 3: The CPU resource that is used by a task will be decremented from its associated server's CPU dynamic attribute q_s, i.e., if a task τ_i executes x time units then $q_s = q_s - x$.
Rule 4: The number of memory requests issued by each task will be decremented from its associated server memory dynamic attribute m_s, i.e., if a task τ_i issues y requests then $m_s = m_s - y$.
Rule 5: A server is in a *suspended state* if any of its CPU- or memory-budget is depleted, i.e., if $m_s = 0$ or $q_s = 0$ then $m_s = q_s = 0$ and the server is suspended until the next server period. Thus, if any of the budgets is depleted then the other remaining budget will be discarded.
Rule 6: We use the idling periodic server strategy [11] for CPU reservation, i.e., if the scheduled server has remaining budget but there is no task ready then it simply idles away its budget until a task becomes ready or the budget depletes.

Note that the MRS follows exactly the rules of the idling server type except for the additional rules related to the memory requests. The memory part of the server behaves like a deferrable server [12] where the capacity of the server is consumed only whenever a memory request is made. The presented rules guarantee that a server only consumes its given CPU budget which limits its effect on the other server that share the same core. The rules also guarantee that a server only consumes its give memory budget, which limits its effect on servers that are located in different cores. In addition and due to the interaction between the two different types of the budgets of the MRS server, the server may overrun its CPU budget. This may happen when a task in a server issues a memory request just before the CPU budget depletion. Since the core is stalling until the memory request is served then the server suspension will be delayed and an overrun can occur. The amount of overrun can be at most equal to the time to serve one memory request. In this paper we ignore such overruns since they are negligible compare to our clock resolution, but for hard real-time analysis the overruns are considered in the analysis of the MRS server [8].

We explain the execution of a MRS using a simple example of a subsystem that consists of five tasks $\tau_1, ..., \tau_5$ where tasks are ordered by their priorities in a descendent order i.e. τ_1 has the highest priority and τ_5 has the lowest priority. Figure 9.2 shows a possible execution scenario of tasks inside a server

$S_s(P_s, Q_s, M_s)$. At the first budget period, we assume that τ_5 is the only ready task and it issues a memory request and then waits until the request is served, and after a very small time τ_4 is released and it preempts the execution of τ_5 when the request is served (during core stalling, no task is allowed to execute) and it issues a memory request. Assume that $M_s = 2$, the memory budget depletes when the request is served and the remaining CPU budget is dropped. As a result the server execution will be suspended until the next budget period. The tasks τ_1 and τ_2 are activated within the first budget period but cannot execute due to the budget expiration of the server. These tasks will get a chance to execute in the next period when the server will be replenished to its full resources. In second budget period, the highest priority ready task τ_1 executes.

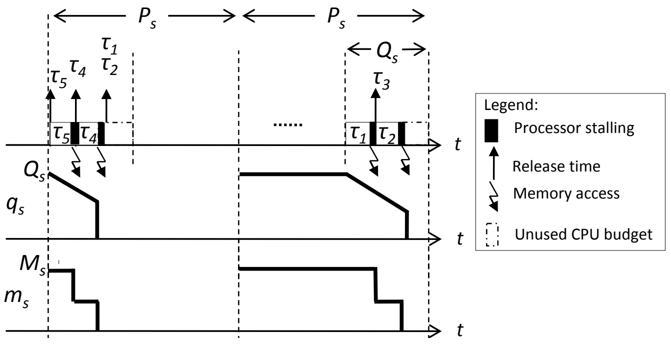


Figure 9.2: An example illustrating the execution of the MRS

The implementation of the periodic server types considering the CPU-budget has been studied extensively (e.g. [13, 14]). However, the main challenge is to add the memory budget and to consider the interactions between the two different types of budgets, more specifically Rule 4 and Rule 5. To implement these additional rules related to the memory part we need to track the memory requests issued by tasks within each server. We will explain this in the following sections:

9.3.2 Determining the consumed memory budget (m_s)

In many cases, a continuous determination and tracking of the consumed memory-bandwidth is very difficult without using a dedicated external hardware that monitors the memory bus. Since we target the use of standard hardware, we use a software-based technique similar to what has been used in [10, 15].

Most modern processors host a range of Performance Monitor Counters (PMCs) which can be used to infer the amount of resources consumed and are used to implement software-based memory throttling. These counters are hardware registers attached with the processor and they contain measures of various programmable events occurring in the processor. Different processor architectures provide different sets of performance counters which makes determination of the consumed memory budget m_s more or less easy and accurate.

9.3.3 Online monitoring and policing of m_s

To provide continuous online monitoring of the consumed memory-bandwidth, we need to continuously monitor counters and store their values. Performance counters are usually programmable and they can be configured to generate an interrupt at overflow, and hence they can be configured to count different types of events. For any given CPU architecture, its usage depends on issues like available events to count, number of available counters (often a small set of counters are available to be programmed to count various events), and the characteristics of the memory-bus.

In our work we implement servers that *enforce/police the consumed bandwidth*. To perform enforcement, an accurate and non-intrusive estimate of the bandwidth consumptions is very important. For policing purposes, using alarms could potentially provide the most accurate approach for accounting of the consumed bandwidth. One method could be to poll the performance monitor counters at each memory-request generation to evaluate m_s. Another method could be to allow an application to generate at most y events before being policed by initializing the event accordingly (Rule 4), in this case an event would be generated after y number of requests. This could obviously result in a situation where $m_s < 0$, which at a first glance would seem inappropriate (or, for hard real-time systems, even dangerous).

9.4 Software framework

The Linux operating system has been selected to be used for the implementation of the multi-resource server approach. The ExSched [14] framework has been used to support hard real-time behavior in Linux. Using this framework, real-time schedulers are developed without the need to patch or modify the main kernel itself. ExSched has been shown to suffer from some overheads. However, for our implementation of a research prototype for realizing the MRS these overheads are acceptable and in our implementation we only focus on the overheads generated by our new MRS functions, not the overhead incurred by the ExSched framework itself.

The ExSched framework supports a user-space library and a loadable kernel module to control the CPU scheduler without modifying the underlying scheduler. The kernel module uses native Linux-kernel primitives and exports them as a simplified interface. Different plug-ins are provided that schedule tasks (user-space applications) using these interfaces. The flow of function calls of user-space applications to the Linux kernel through the core module are described in detail in [14]. New plug-ins can be developed by extending the scheduling policies, for example two hierarchical scheduler plug-ins (for fixed-priority scheduling and for EDF scheduling) and three multi-core scheduler plug-ins are presented in [14]. Since we use hierarchial scheduling for the MRS, we explain below how a two-level fixed-priority hierarchial scheduler is implemented in ExSched.

The uni-core hierarchical scheduler plug-in supports a two-level hierarchical scheduler and it schedules tasks within their servers [14]. All tasks are initially migrated to core 0 and they are assigned to their specific servers at system start using `job_init_plugin`. Tasks are executed periodically using the `rt_wait_for_period` API that internally calls the `job_complete_plugin` and `job_release_plugin` interfaces. Two main interrupt-handler functions to handle the server's activation and depletion activities are `server_release_handler` and `server_complete_handler`, respectively. They are triggered by server release and deplete events through timer activations. These functions release a server (along with its tasks) with its full CPU-budget to execute and suspend the server at its CPU-budget depletion (along with its tasks) respectively. A server ready queue and a server release queue are implemented using bitmaps (as Linux 2.6 native task ready-queue) to store the ready and depleted servers respectively.

9.5 MRS implementation

The implementation details of the MRS in the context of partitioned multi-core scheduling are presented here. The hierarchical scheduler implemented in the ExSched framework manages the CPU-budget using FPPS at both levels of scheduling [14]. We extend the hierarchical scheduler to support the memory-budget and we extend it for multi-core platforms by implementing partitioned scheduling.

9.5.1 MRS design for partitioned HSF

Global structures: To implement a partitioned multi-core hierarchical scheduler, an `SERVERS[]` array of `server_struct` type, and an `per_CPU[NR_RT_CPUS]` array of `perCPU_struct` type are used in the system globally as depicted in Figure 9.3. All other structures including timers and queues for servers are maintained per core. The `SERVERS[]` array holds all servers in the system. The only reason why the global variable `SERVERS[]` is maintained as an implementation choice is that a user API to create servers will be much easier and also to preserve the API scheme used by ExSched.

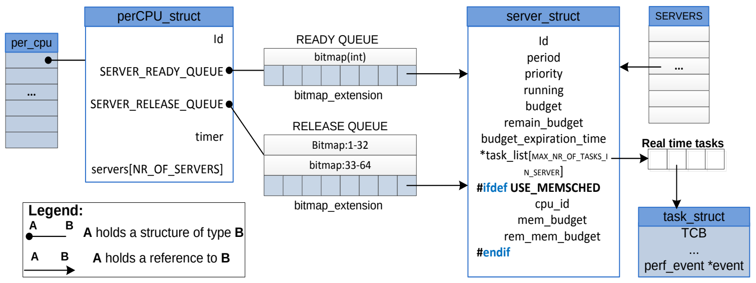


Figure 9.3: MRS design for partitioned HSF

`perCPU_struct` structure: This structure contains core id, a timer and two queues, a `SERVER_READY_QUEUE` and a `SERVER_RELEASE_QUEUE`, to schedule servers on that core. Both server-queues are of bitmap extension type arrays of pointers. The timer is used to activate the server events on that core[2]. A server can be either in `SERVER_READY_QUEUE` or `SERVER_RELEASE_QUEUE` at any time, and is implied as ready (Rule 2) or inactive (Rule 5) respectively. Only one highest priority servers from the `SERVER_READY_QUEUE` executes at a time on each core. The `perCPU_struct` also contains an `severs[NR_OF_SERVERS]` array that is used for mapping servers to a specific core, and it stores all servers' IDs that are allocated to that core. This local `severs` array's index is mapped to the global `SERVERS[]` array's index for a faster access of server parameters during decisions making like comparing priority, updating remaining budget, referring to period etc.

Server control block: This contains all information needed by an MRS server in a `server_struct`, i.e. the `period` (P_s), `priority`, `budget` (Q_s), `remaining_`

[2]More details on queues and timers can be found in code or in [14].

`budget` (q_s), `budget_expiration_time` and a `task_list` that points to tasks belonging to the server as presented in Figure 9.3. To execute the server on a particular core, the `CPU_id` variable is added to the `server_struct`. Further, the `mem_budget` (M_s) and `remain_mem_budget` (m_s) variables are added to monitor the memory-bandwidth consumption of MRS.

Server release and complete handlers: The two timer event handlers used to control activation and deactivation of servers in the multi-core HSF are `server_release_handler()` and `server_complete_handler()`, respectively. These handlers are triggered when previously setup timer events expire due to periodic activation, budget depletion or pre-emption by a higher priority server. Multiple activities are performed in these handlers such as budget updating, task enqueuing/dequeuing, new timer setup etc. More details can be found in code.

9.5.2 Implementing memory throttling by configuring and accessing counters

On multi-core processors each core usually contains its own set of hardware performance counters making it possible to account for memory events happening on a given processor. We implement the performance counters by using the `perf_event` interface of the Linux kernel. We use this interface within our implementation, and create and install performance counter events in the PMU[3]. To account for the L2 misses incurred by a specific server, several reservation schemes can be used, namely per-task or per-core assignment. Since multiple servers execute on each core, the per-core assignment scheme does not suit our problem. Therefore, we configure the counter for the per-task scheme, thereby accounting for events for only those tasks that belong to the server. In order to save and restore the counter registers upon a task context-switch, a `struct perf_event * event` structure is added to the task control block as shown in Figure 9.3.

A memory-requests counting event is created using `struct perf_event *init_counter (task_struct * task, int cpu)` API. Since tasks are statically allocated to a specific core, the event is also bound with the task and the core. It is configured to monitor hardware events (`PERF_TYPE_HARDWARE`) and to measure the L2 misses (`PERF_COUNT_HW_CACHE_MISSES`) in the kernel space. The counting event is created when a task calls its `task_run` function at its initialization and connects to its server.

[3]Performance Monitoring Unit (PMU) in the Intel architecture where performance counters are implemented.

9.5.3 Online monitoring and policing of the memory budget

For online monitoring of the memory-budget, the performance counter is configured to cause an interrupt at an overflow. The `sample_period` is set to 1 to call the overflow handler `memory_overflow_handler()` at each memory-request. The memory budget of the event-generating tasks' server is decremented in the handler, i.e. m_s--; `remain_mem_budget` variable of each servers `server_struct` is used to monitor the consumption.

At memory-budget exhaustion (Rule 5), the `memory_server_complete_handler()` is called to enforce the server depletion. This handler works in the same manner as of `server_complete_handler()`, except that it is not an interrupt handler itself, rather it is called from the interrupt handler. It sets up the next activation time of the depleted server, it deactivates/dequeues the server along with its task set, and it activates/enqueues the next highest priority server with its tasks. If no server is ready at that time, then the idle tasks or other low priority tasks of Linux will execute.

9.6 Evaluation setup

9.6.1 Hardware and software platforms

All experiments are performed on an Intel core 2 CPU 6700 with two cores running at a frequency of 2.66 GHz having 32+32KB of local L1 instruction- and data-cache, and sharing an L2 cache of size 4MB. The frequency scaling is disabled to prevent the system from going into power-save modes and reducing its clock-frequency.

We use Ubuntu 10.04.4 LTS with Linux kernel version 3.6.0-rt. The scheduler resolution (system tick) is set to $1ms$. The standard C library is used for programming and all programs are compiled using the gcc compiler.

9.6.2 The behavior of synthetic tasks

Two different synthetic task-types are used in the case study and in the synthetic evaluations, namely *normal task*, and *memory intensive task*, their behavior is described here:

The normal task generates a relatively low number of requests per server period as compared to the memory intensive task. The task's code iterates a dynamic linked list consisting of a total of 140000 nodes (each node is of 8 bytes, and the size of the list is approx. 1MB) and it assigns an integer value to the single data item of the list. The WCET of the task is dependent on the selected number of iterations. We measured that on our platform, traversing 140000 nodes once as a single task in the system takes approximately $1ms$ to execute. For example if a task has an execution time of $2ms$ then it will iterate the list 2 times, so that the normal task could iterate the list for a time that is close enough to its WCET. In our investigation, the use of this dynamic linked list generates a good amount of reads and writes to the memory. Some accesses goes to the cache (due to good locality of consecutively allocated memory blocks) but we also get a quite large amount of cache misses, resulting in memory requests on the shared bus.

The memory intensive task generates a very high number of requests per server period. The task iterates through the same kind of linked list as the normal task except that the number of nodes in the list is increased by 4 times. Consequently, the list size (list size is slightly greater than 4MB) becomes much bigger than the list size of a normal task. Further, the task is executed continuously (i.e. it never goes idle waiting for a new period) within a server, thus the task is only bounded by its server's reservation and it will execute as much as the server allows it to. Hence, this task will heavily affect other tasks' execution in the system due to its unbounded execution time and a very high memory-bandwidth usage.

9.7 Case study: Executing a legacy application

The purpose of this experiment is (1) to show that a legacy application that works well when executed on one core may fail to deliver its service if applications on other cores consume too much resources, and (2) to show that if applications resource utilization are bounded with the MRS, then we can protect the legacy application and allow it to deliver its service.

We use a soft real-time legacy application: *mplayer*[4], that decodes and plays an audio/video file and it requires continuous access to memory to fetch and process video frames. mplayer demands a high amount of memory bandwidth to display the video at an acceptable rate. Further, the timing is important for mplayer, otherwise it starts dropping video frames affecting the quality of service. We execute mplayer as a task within a server on core 0 that is bounded only by its server's CPU reservation. On the other core, we execute synthetic tasks within servers.

For the case study we have executed a high-definition HD video, i.e. a trailer of Avatar ([H264] 1920x800 24bpp) of a total of 260 seconds dura-

[4]http://www.mplayerhq.hu

Server	Core	Priority	Period	CPU-budget
`mplayer`	0	`High`	15	10
`Server0`	1	`High`	80	12
`Server1`	1	`Low`	80	12

Table 9.1: The servers' specification for the case study.

tion. To assess the performance of the quality of service delivered by mplayer, we use the *number of dropped frames* as our benchmark. The servers used for the case study are presented in Table 9.1. The case study is performed in three steps, presented below. In these steps, the servers' priority, period, and CPU-budget remain the same as given in Table 9.1, while the tasks' behaviour and the memory-budget vary in different experiments. Server and task period, CPU-budget, and Worst Case Execution Time (WCET) values are presented in ms, while the memory-budget is provided as a number of memory-requests. Note that the memory-bandwidth usage can be easily calculated by multiplying the number of memory requests to the cache-line size (64 bytes in our platform). The details for these steps are presented here:

(1): We executed mplayer with our example video-file as a stand-alone application on core 0 to find its normal execution behavior having all resources available. We found that it drops 0% of the frames while playing the video at a rate of $25fps$. These measures are later compared when mplayer is executed along with other MRSs in the system and the resources are shared among all applications/subsystems.

(2): We inserted two MRSs on core 1, each executing two tasks as given in Table 9.2. Note that a higher number means a higher priority for tasks. Without memory reservation on MRSs, the mplayer dropped 1% of the frames due to the global interference. However, mplayer executed with 0% dropped frames when the MRSs on core 1 are throttled with a memory-budget of 1100. Hence, using MRSs, mplayer can be executed with desired results, which was not possible without MRS.

(3): We introduced heavier memory-traffic by executing two memory intensive tasks, where each server on core 1 is executing one task. As mentioned previously, both tasks execute continuously, bounded by their server's CPU-budgets respectively, and produce a heavy memory traffic.

Executing the system without memory reservation on MRSs produce a bad effect due to a global interference on mplayer by dropping 5% of the frames. This effect is significantly reduced by throttling two MRSs on core 1: with

Task	Server	Priority	Period	WCET
`Task1`	`Server0`	98	160	10
`Task2`	`Server0`	97	160	14
`Task3`	`Server1`	98	200	8
`Task4`	`Server1`	97	200	8

Table 9.2: Tasks properties and their assignment to servers.

a memory-budget of 750 requests, the dropped frames decreased to 3%; and with a memory-budget of 200, the dropped frames decreased to 0.3% (only 17 frames dropped from a total of 5013 frames). Hence, using MRSs, mplayer can be executed with limited and acceptable effect on its performance, which was not possible without MRS. This case study shows that a predictable execution of a legacy uni-core application can be achieved on a multi-core platform by providing both temporal- and memory-bandwidth isolations through the usage of MRSs.

When running the memory intensive task-behavior on core 1 we see a slight decrease in the performance of mplayer compared to the normal task-behavior. While we have not investigated the reason for this decrease in detail, we hypothesize that the reason is related to increased cache-pollution in the shared L2 cache – making the mplayer experience more cache-misses and thus performing slightly worse. In the next section we show that cache-pollution *is* an issue that matters and that it can cause temporal interference among tasks.

9.8 Synthetic evaluation – Results and analysis

Here, we measure performance overheads of the implementation and we evaluate the timing isolation of the MRS using a set of synthetic tasks.

9.8.1 Performance assessments

We present the overheads for memory related functionality of the MRS. The first measured overhead is of executing the Performance Monitor Counters (PMC) and it is negligible as it only writes to a register of a core. The overhead of the interrupt function to handle overflow `memory_interrupt()` is $56ns$ (nano seconds) on average. This interrupt is called at each memory-request. We observed a maximum of 130 requests during $1ms$, that means in worst case 0.7% overhead for our experiments.

Other overhead measures are the time required to execute (1) the `server_release_handler()` function that activates servers and its tasks at the server's activation time, (2) the `server_complete_handler()` that suspends servers and its tasks at server's CPU-budget depletion, and finally (3) the `memory_server_complete_handler()` that suspends servers and its tasks at server's memory-budget depletion. Two scenarios are accounted for each of these functions: first, the function is called when *no other active server* was on the core (the idle task was executing) and a server context-switch will occur to execute the newly released server; and second, another *active server* was executing on the core, in this case a server context-switch may occur to execute the newly released server depending upon the server's priority. The overhead of a server context-switch is included within the measures.

The system is executed for 5 minutes and overhead measures are extracted for each scenario as presented in Figure 9.4. The *Count* column in the table represents the total number of times that a particular scenario executed and then average, minimum, maximum, and standard deviation on these values are calculated and presented. All values are given in micro-seconds (μs). It is obvious from the table that overheads are very low, i.e no more than 0.68% for all functions for our experiments. The total overheads of the system are high for the underlying ExSched framework due to having a kernel modification-free solution and these overheads are presented in [14].

Scenarios	server_release_handler()					server_complete_handler()					memory_server_complete_handler()				
	Count	Avg.	Min.	Max.	St. Dev.	Count	Avg.	Min.	Max.	St. Dev.	Count	Avg.	Min.	Max.	St. Dev.
No other active server	7443	5.9758	2	17	1.9047	6025	6.0211	2	21	2.8607	1239	5.5343	4	7	0.5180
Another active server	2478	7.3010	5	15	0.7192	11125	6.895	3	17	1.3890	1250	5.529	4	12	0.7557

Figure 9.4: Overhead measures for the memory related functionality of the MRS.

9.8.2 Synthetic experiments

Synthetic experiments are performed by executing a schedulable example consisting of five servers along with their task sets on both cores for 10 seconds. Two servers, i.e. `Server0` and `Server1` are executed on core 0, while all other servers are executed on core 1. The servers' timing properties and their assignment to the CPU-core is given in Table 9.3. Tasks' properties and their assignment to their corresponding server is given in Table 9.4. To execute our tasks before the Linux tasks and to avoid task pre-emptions due to other Linux tasks, we have assigned the highest priority values to tasks.

Server	Core	Priority	Period	CPU-budget	Memory-budget
`Server0`	0	`High`	24	8	650
`Server1`	0	`Low`	40	16	750
`Server2`	1	`Medium`	40	8	900
`Server3`	1	`High`	80	12	1100
`Server4`	1	`Low`	80	12	1100

Table 9.3: The servers' specification to test the behaviors.

Task	Server	Priority	Period	WCET
`Task1`	`Server0`	98	40	2
`Task2`	`Server0`	97	48	4
`Task3`	`Server1`	98	60	8
`Task4`	`Server2`	98	60	4
`Task5`	`Server2`	97	160	10
`Task6`	`Server3`	97	160	14
`Task7`	`Server4`	98	200	8
`Task8`	`Server4`	97	200	8

Table 9.4: Tasks properties and their assignment to servers.

All synthetic experiments are performed in two steps: first executing all servers and tasks without memory reservation using a simple idling periodic server; and then executing using memory reservation as the MRS. The number of missed deadlines for all tasks are measured for both steps to examine the effect of global interference and to reveal the memory reservation and performance isolation properties of the MRS.

Experiment 1: Memory-bandwidth reservation

This experiment is performed to illustrate the memory-bandwidth reservation of MRS in the context of a schedulable system by means of a trace of execution and by calculating the number of missed deadlines for all tasks. To fully utilize the CPU- and memory-resources, we execute the server and task sets described in Section 9.8.2 on both cores using only normal tasks in the servers. Each experiment is executed in two steps and the total sum of missed deadlines for all tasks is measured. Since normal tasks are low memory-intensive and they require a low number of resources, they get a good chance to execute and thereby never miss their deadlines.

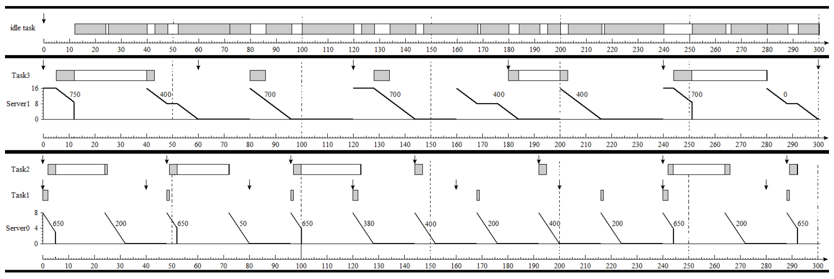


Figure 9.5: Memory-bandwidth reservation of MRS: an execution trace of core 0 - using the normal tasks

The visualization of the execution for the MRS on core 0 is presented in Figure 9.5. The execution trace of core 1 shows the same behavior; we omit it due to limitation of space in the paper. In the diagram, the horizontal axis represents the time in ms starting from 0. In the task's visualization, the arrow represents task arrival, a gray rectangle means task execution, a white rectangle represents either a local pre-emption by another task in the same server or a global pre-emption due to its server's budget depletion or its server's pre-emption by a higher priority server. In the server's visualization, the numbers along the vertical axis are the server's CPU-capacity and the number along the diagonal line represents the memory-capacity (or the number of requests made by the server) during the period. The diagonal line represents the server execution, the vertical line shows the server depletion due to memory-budget, while the horizontal line represents either the waiting time for the next activation (when the budget has depleted) or the waiting for its turn to execute (when some other higher priority server is executing). There is one idle task per core that executes only when no task is ready on the core.

Note that it is clear from the diagram that all servers and tasks execute smoothly because of enough available resources. At the start of the system, both servers deplete due to their memory-budget exhaustion since the caches are empty and the system fetches a lot of data during this time to properly start the execution. This effect has been observed in all experiments.

Experiment 2: Performance isolation effect of memory reservation

This experiment is performed to illustrate the memory-isolation effect among the MRSs due to memory bandwidth reservation, even during the overload situation. For example, if one MRS is overloaded and its tasks miss-behave, produce a large number of memory requests, and fill-up the memory-bandwidth, it should not affect the execution of other MRSs in the system.

For this purpose, all servers execute the normal tasks except `Server1` that executes the memory intensive task as `Task3` for longer duration and produces an increased number of memory-requests. The experiment is executed without- and with memory reservation steps and traces of execution for both steps for core 0 are presented in Figures 9.6 and 9.7 respectively. The total sum of deadline misses (`No.of DMisses`) during the total number of task's activation (`Tot. Activations`) by all tasks is also measured and is presented in Table 9.5.

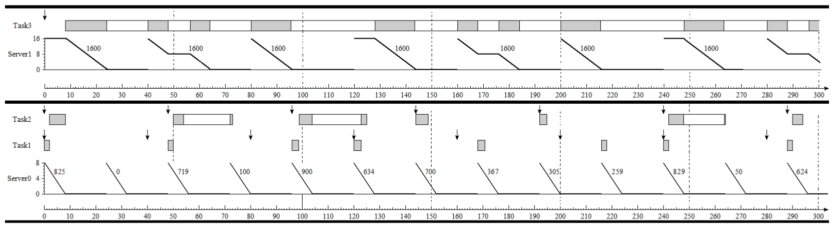


Figure 9.6: Execution trace without memory reservation on Core 0 - using the memory intensive tasks

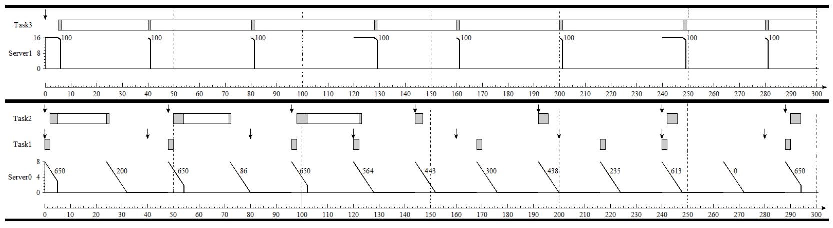


Figure 9.7: Trace showing temporal- and memory-isolation among MRSs - using the memory intensive tasks

`Server1` is over-flooding the system with its memory-bandwidth usage thereby highly affecting the execution of all other tasks in the system when executed without memory-throttling as obvious from looking at Figure 9.6 and from the number of deadline-misses outlined in Table 9.5. (`DMisses` present total deadline misses out of total number of task activations `Act.`) Tasks of other servers miss their deadlines due to the reduced availability of memory-bandwidth. Both the local and the global interference has been observed here. However, when the same setup is executed by enabling memory reservation in the MRS, `Server1` gets bounded by its memory-budget and its overloaded memory-bandwidth usage keeps on reducing its affect on the execution behavior of other servers and tasks in the system, and finally at the memory-budget of 100 requests per period, the number of deadline-misses become 0 for all normal tasks as obvious from the forth column of Table 9.5.

Note that when executed without memory reservation, the execution of tasks of `Server0` becomes unpredictable. Not only the tasks' executions times are increased a lot but they are also generating a varying number of memory-requests, as obvious from Figure 9.6. Due to space reasons, the idle task execution is removed from the figure. Since the memory-bandwidth requirement from normal tasks is not as high as from memory intensive task, we consider that it is the bad effect of not only the high bandwidth usage but also of the cache pollution from `Task3`. However, we observe from the detailed execution trace of Figure 9.7 with memory reservation that the execution times of the normal tasks are slightly increased in very few periods. Mostly the trace of `Server0` and its tasks resemble the trace of experiment 1 with normal tasks in Figure 9.5. This could be due to the cache pollution effect.The use of the MRS highly reduces the cache pollution by limiting the high-demanding bandwidth server, but it could not delete it completely.

Tasks	Without mem. reserv.		With mem. reserv.		throttle all except `server1`	
	`DMisses`	`Act.`	`DMisses`	`Act.`	`DMisses`	`Act.`
`Task1`	0	249	0	236	0	238
`Task2`	1	206	0	197	0	198
`Task4`	0	166	0	157	3	155
`Task5`	22	35	0	59	24	29
`Task6`	2	60	0	59	24	34
`Task7`	0	50	0	47	0	48
`Task8`	1	48	0	47	0	47

Table 9.5: Comparison of deadline misses by tasks to evaluate the behavior of memory-throttling.

To further investigate the effect of `Server1` on other servers in the system and to confirm that the bad-effect on tasks' executions is only due to this server, we perform a third step by reserving all servers for memory except `Server1` and we measured the deadline miss count. As it is clear from the sixth column of Table 9.5, tasks suffer from the global interference and cache pollution and they miss their deadlines due to the un-bounded amount of memory requests from `Server1`.

Experiment 3: Reducing cache pollution by memory reservation

To further investigate the cache pollution effect and its reduction due to the memory reservation, we execute a *cache polluting task* in `Server1`. All other servers and tasks have the same specifications and they execute the same code as presented in Experiment 1.

The *cache polluting task* is designed to examine the cache pollution effects and it is executed continuously in a server like the memory intensive task. However, its code is modified in two ways: first the size of the linked list is now multiplied by 10 to make the cache polluted; secondly, it dynamically creates nodes for two linked lists, reads the data from the first list and writes to the second list, and then deletes the nodes. Note that the caches have limited effect in this case of extreme cache pollution since the data size is much bigger than the cache size and additionally it is constantly changing over a chunk of memory due to allocations and de-allocations of memory in the same iteration. Therefore, this task represents the worst case of memory access pattern.

We observed that without memory-throttling, all tasks miss their deadlines many times as expected. However when we executed the experiment using MRSs with memory-throttling, there were always either two or three different tasks missing their deadlines once per execution, hence either two or three deadlines were always missed (due to space reasons we are not presenting all the data here). This experiment shows that our solution has the potential to reduce the cache pollution problem, however not solving it completely.

9.9 Related work

The problem of contention of shared resources has gained a significant importance in the context of multi-core embedded systems. Software-based partitioning is one technique to provide predictable execution. In avionics, ARINC-653 is used as a platform to implement partitioned software with emphasis on predictability and safety-critical issues [16]. It provides fully deterministic top-level Time Division Multiple Access (TDMA) based schedule for unicore platform. Some highly predictable TDMA based techniques are used to access the shared resources (memory bus arbitration) using a multiprocessor systems-on-a-chip (SoC) architecture. Rosen et al. [17] measured the effects of cache misses on the shared bus traffic where the memory accesses are confined at the beginning and at the end of the tasks. Later Schranzhofer et al. [18] relaxed the assumption of fixed positions for the bus access by arbitrating the shared bus. TDMA arbitration techniques eliminate the interference of other tasks due to accessing shared resources through isolation; however, they are limited in the usage of only a specified hardware. Akesson et al. [19] proposed a two-step approach to share a predictable SDRAM memory controller for real-time systems. This is a key component in CoMPSoC [20]. Stuijk et al. [21] used Synchronous Dataflow Graphs (SDFG) for allocating resources on a heterogeneous multi-processor system and provide throughput guarantees. Zimmer et al. [22] provides a TDMA-like approach that optimally maps tasks on network-on-chip (NoC) by implementing a heuristic-based solver. The research is also going on for using COTS microprocessors and systems-on-a-chip (SoC) in complex and safety-critical avionics with the main focus to identify risks of using SoCs, and how to support the certification of aircraft [23]. Our approach, however, uses SMP COTS hardware and it is software based using performance counters which are available in almost all processors.

Pellizzoni et al. [24] initially proposed the division of tasks into superblock sets by managing most of the memory request either at the start or at the end of the execution blocks. This idea of superblocks was later used in TDMA arbitration [18]. Bak et al. presented a memory aware scheduling for multi-core systems in [25]. They use PRedictable Execution Model (PREM) [26] compliant task sets for their simulation-based evaluations. However, PREM requires modifications in the existing code, hence this approach is not compliant with our goal to execute legacy systems on the multi-core platform.

Some approaches to WCET analysis are emerging which analyze memory-bus contention, e.g. [27]. However, WCET-approaches do not tackle system wide issues and do not give any direct support to provide isolation between subsystems. Schliecker et al. [6] have presented a method to bound the shared resource load by computing the maximum number of consecutive cache misses generated during a specified time interval. The joint bound is presented for a set of tasks executing on the same core covering the effects of both intrinsic and pre-emption related cache misses. A tighter upper bound on the number of requests is presented by Dasari et al. [7] where they solve the problem of interleaving cache effects by using non-preemptive task scheduling. They have used PMCs in the Intel platform running the VxWorks operating system to measure the number of requests that can be issued by a task. However, these works lack the consideration of shared memory-bandwidth and the use of memory servers to limit the access to memory-bandwidth.

Recently a server-based approach to bound the memory load of low priority non-critical tasks executing on non-critical cores was presented by Yun et al. in [10] for an Intel architecture running Linux. In their model, one memory server is implemented on each non-critical core to limit memory requests generated by tasks that are located on that core. Hence the interference from other non-critical cores on the critical core is bounded. The servers are implemented on Linux using cgroups in [10]. This approach might not be suitable for those real systems that may contain more than one critical application. In addition, using one memory server in each non-critical core will degrade the performance of all applications in that core even if the core contains only one memory intensive task. This work has been extended in [15] by using a memory reclaiming technique when a core is not fully utilizing its allocated memory budget, and is implemented as a dynamic loadable Linux kernel module with some small modifications in the main kernel.

We propose a more general approach by implementing the MRS that handles both time and memory aspects reserved resources. Multiple subsystem/ applications can share one core through multiple MRS's. Our memory throttling mechanism is proposed per server level instead of per core level (as in [10, 15]) and thereby the time and memory reservation aspects are applied per server.

9.10 Conclusions

We have presented the first implementation the Multi-Resource Server (MRS) for reserving both CPU- and memory-bandwidth for multi-core systems. An evaluation shows that overhead of our implemented functionality is low. Furthermore, we have demonstrated the MRS suitability to execute legacy unicore applications in a predictable manner on a multi-core platform by providing temporal isolation both between applications running on the same core and between applications running on different cores.

Our demonstration shows that scheduling alone (i.e. controlling the allocation of resources over time) is not enough to achieve complete timing isolation. We observe that cache-pollution can have a tangible effect on timing properties of tasks executing in different serves. However, we also show that MRS, itself, can be used to mitigate cache-pollution since it bounds the effect on the shared cache for each server. Nevertheless, we conclude that our MRS should be complemented with some technique to remove/bound cache-pollution amongst servers, e.g., accounting for it in the analysis [6, 7] or implementing a cache partitioning solution [4, 5].

Another future direction is to find an algorithm to calculate the optimum budgets for both resources of the MRS. Some smart online algorithms can be developed to assign the unused capacity of one resource to another server to improve overall average response times. We also look at implementing the MRS without ExSched to achieve better performance, as ExSched requires some overhead by itself.

Bibliography

[1] J. P. Lehoczky, L. Sha, and J.K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proc. 8th IEEE Real-Time Systems Symposium (RTSS' 87)*, pages 261–270, December 1987.

[2] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems*, 1(1):27–60, June 1989.

[3] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proc. 24th IEEE Real-Time Systems Symposium (RTSS' 03)*, pages 2–13, December 2003.

[4] B.C. Ward, J.L. Herman, C.J. Kenna, and J.H. Anderson. Making shared caches more predictable on multicore platforms. In *Proc. of the 24th Euromicro Conf. on Real-Time Systems (ECRTS' 13)*, pages 157–167, July 2013.

[5] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *Proc. 19th IEEE Real-Time Technology and Applications Symposium (RTAS' 13)*, pages 45–54, April 2013.

[6] S. Schliecker and R. Ernst. Real-time Performance Analysis of Multiprocessor Systems with Shared Memory. *ACM Transactions in Embedded Computing Systems*, 10(2):22:1–22:27, January 2011.

[7] D. Dasari and B. Anderssom and V. Nelis and S.M. Petters and A. Easwaran and L. Jinkyu . Response Time Analysis of COTS-Based Multicores Considering the Contention on the Shared Memory Bus. In *Proc. of the IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom '11)*, pages 1068 – 1075, November 2011.

[8] M. Behnam, R. Inam, T. Nolte, and M. Sjödin. Multi-core composability in the face of memory bus contention. *ACM SIGBED Review, Special Issue on 5th Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, 10(3), 2013.

[9] I. Shin and I. Lee. Periodic Resource Model for Compositional Real-Time Guarantees. In *Proc. 24th IEEE Real-Time Systems Symposium (RTSS' 03)*, pages 2–13, December 2003.

[10] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory- access Control in Multiprocessor for Real-Time Systems with Mixed Criticality. In *Proc. of the 24th Euromicro Conf. on Real-Time Systems (ECRTS' 12)*, July 2012.

[11] L. Sha, J.P. Lehoczky, and R. Rajkumar. Solutions for some Practical problems in Prioritised Preemptive Scheduling. In *Proc. 7th IEEE Real-Time Systems Symposium (RTSS' 86)*, pages 181–191, December 1986.

[12] 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.

[13] R. Inam, J. Mäki-Turja, M. Sjödin, S. M. H. Ashjaei, and S. Afshar. Support for Hierarchical Scheduling in FreeRTOS. In *16th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA' 11)*, pages 1–10, 2011.

[14] M. Åsberg, T. Nolte, S. Kato, and R. Rajkumar. ExSched: an external cpu scheduler framework for real-time systems. In *18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA' 12)*, pages 240–249, August 2012.

[15] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Proc. 19th IEEE Real-Time Technology and Applications Symposium (RTAS' 13)*, pages 55–64, 2013.

[16] S. Han and H.W. Jin. Full virtualization based ARINC - 653 partitioning. In *30th IEEE/AIAA Digital Avionics Systems Conference*, pages 7E1–1–7E1–11, October 2011.

[17] J. Rosen, A. Andrei, P. Eles, and Z. Peng. Bus Access Optimization for Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. In *Proc. 28th IEEE Real-Time Systems Symposium (RTSS' 07)*, pages 49–60, December 2007.

[18] A. Schranzhofer, R. Pellizzoni, J.-J. Chen, L. Thiele, and M. Caccamo. Worst-case Response Time Analysis of Resource Access Models in Multi-core Systems. In *Proc. of the 47th Design Automation Conference (DAC '10)*, pages 332–337. ACM, 2010.

[19] B. Akesson, K. Goossens, and M. Ringhofer. Predator: A Predictable SDRAM Memory Controller. In *Int'l Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 251–256, September 2007.

[20] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken. CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM Trans. Des. Autom. Electron. Syst.*, 14(1):2:1–2:24, January 2009.

[21] S. Stuijk, T. Basten, M. C W Geilen, and H. Corporaal. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *44th ACM/IEEE Design Automation Conference (DAC '07)*, pages 777–782, June 2007.

[22] C. Zimmer and F. Mueller. Low contention mapping of real-time tasks onto TilePro 64 core processors. In *Proc. 18th IEEE Real-Time Technology and Applications Symposium (RTAS' 12)*, pages 131–140, April 2012.

[23] Microprocessor evaluations for safety-critical, real-time applications: Authority for expenditure no. 43 phase 5 report, May 2011. http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/media/11-5.pdf.

[24] R. Pellizzoni and A. Schranzhofer and J.-J.Chen and M. Caccamo and L. Thiele. Worst Case Delay Analysis for Memory Interference in Multicore Systems. In *Proc. of the Conference on Design, Automation and Test in Europe (DATE' 10)*, pages 759–764, 2010.

[25] S. Bak and G. Yao and R. Pellizzoni and M. Caccamo. Memory-Aware Scheduling of Multicore Task Sets for Real-Time Systems. In *Proc. of the*

IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '12), 2012.

[26] R. Pellizzoni and E. Betti and S. Bak and G. Yao and J. Criswell and M. Caccamo and R. Kegley. A PRedictable Execution Model for Cots-based Embedded Systems. In *Proc. of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS' 11)*, 2011.

[27] T. Kelter and H. Falk and P. Marwedel and S. Chattopadhyay and A. Roychoudhury. Bus-Aware Multicore WCET Analysis Through TDMA Offset Bounds. In *Proc. of the 23th Euromicro Conf. on Real-Time Systems (ECRTS' 11)*, pages 3 – 12, June 2011.

Chapter 10

Paper E: Worst case delay analysis of a DRAM memory request for COTS multicore architectures

Rafia Inam, Moris Behnam, Mikael Sjödin

In Seventh Swedish Workshop on Multicore Computing (MCC 14), Lund, Sweden, November, 2014.

Abstract

Dynamic RAM (DRAM) is a source of memory contention and interference problems on commercial of the shelf (COTS) multicore architectures. Due to its variable access time, it can greatly influence the task's WCET and can lead to unpredictability. In this paper, we provide a worst case delay analysis for a DRAM memory request to safely bound memory contention on multicore architectures. We derive a worst-case service time for a single memory request and then combine it with the per-request memory interference that can be generated by the tasks executing on same or different cores in order to generate the delay bound.

10.1 Introduction

The real-time applications that are executed concurrently on COTS multicore platforms, face the new challenges due to sharing multiple different physical resources including CPU, shared caches, memory bandwidth and memory. Contention for the shared physical resources is a natural consequence of sharing [1]. It does not only reduce throughput but also affects the predictability of real-time applications.

Modern multicore architectures use a single-port Double Data Rate Dynamic RAM (DDR DRAM) as their main memory resource [2], which is shared among all cores. It is becoming a significant source of memory contention and interference problems that lead to unpredictability. It exhibits a highly variable DRAM access-time. Multiple studies provide bounds on memory interference delay by considering a constant access time [3, 4], and a variable access time [5, 6] for tasks executing concurrently on different cores and contending for memory accesses. Many hardware-based solutions have been proposed to eliminate these limitations at the level of DRAM controller [7, 8]. However, the real-time applications developed for COTS hardware cannot use this specialized hardware.

We provide a worst case delay analysis for a DRAM memory request to safely bound memory contention for multicore architectures. First, a worst-case service time for a single memory request is derived considering the worst case latency scenarios of DRAM commands. The service time is then combined with the per-request memory interference that can be generated by the tasks executing on same or different cores to generate the delay bound. Our analysis is similar to the work of [6], except that we have added additional constraints for shared memory banks (details in Section 10.4).

Section 10.2 provides the background on DRAM. Section 10.3 explains our system model. Memory interference delay analysis is presented in Section 10.4 and finally Section 10.5 concludes the paper.

10.2 DRAM background

A DRAM memory system consists of a DRAM controller and a memory device as shown in Figure 10.1. The controller serves the memory requests (i.e. schedules memory requests generated by CPU and sent to DRAM) and the memory device stores the actual data. The memory device consists of ranks and only one rank is accessed at a time. Each rank consists of multiple DRAM

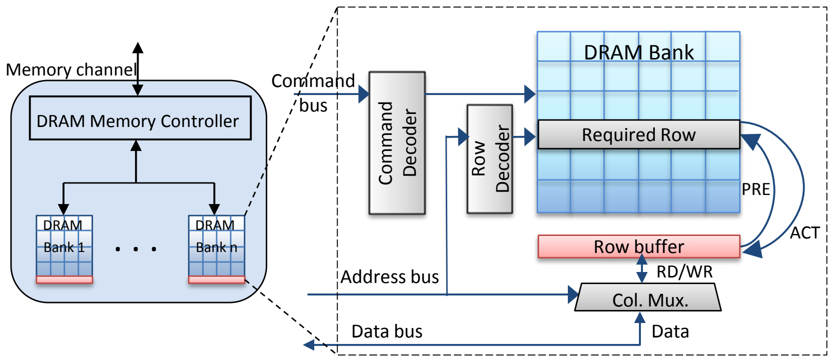


Figure 10.1: DDR DRAM banks : organization and functionality

banks that can be accessed independently. The memory requests to different banks can be served concurrently, this is called *Bank-Level Parallelism (BLP)*. Only one of them can transfer data at a particular time on the data bus. [1]

Each bank consists of a *row-buffer*, and a two-dimensional structure of rows and columns of DRAM cells that actually stores data, as depicted in Figure 10.1. The data of a bank can only be accessed from the row-buffer of the bank. Thus to access the data from a bank, first the required row is cached into the row-buffer using the row decoder, and then the data is accessed from that row. The data of the particular column is accessed (read from and written to) from the row-buffer using the column multiplexer as shown in Figure 10.1. A column represents a small set of consecutive bits within a row. Thus the row-buffer serves as a buffer for the last-accessed request. All the subsequent requests to the columns of the same row do not require caching the row into the buffer, and are directly performed by accessing the required column of the row-buffer for faster access. The row that is cached is called an *open row*. A request to an open row is considered as a *row-hit*. If the currently opened row is different than the requested row, then first the opened row is saved and then the requested row is fetched into the row-buffer; it is called a *row-conflict*.

DRAM controller performs *internal scheduling algorithms* to re-order memory requests in order to improve the row-hit ratio and to maximize the overall throughput [9]. Since the row-hit latency is much less than the row-conflict latency, the DRAM controller prefers the row-hit request over the row-conflict requests, thus it unfairly prioritizes threads with high row-buffer locality. It

[1]This is similar for rank level parallelism. We consider one rank and one channel in this work.

schedules memory requests using *First-Ready First Come First Served (FR-FCFS)* algorithm [10] that prioritizes the ready DRAM commands (row-hit memory request) over others and for ties it prioritizes older requests. It means that the memory requests arriving earlier may be serviced later than ones arriving later in the memory system. FR-FCFS works better with the *open row policy* that keeps the row-buffer open rather than *close row policy* that closes the row-buffer after serving each request.

The following commands are used to access the data from a bank (see Figure 10.1): Activate command (ACT) loads the requested row into the row-buffer using the row decoder; Precharge (PRE) writes back the currently opened row; (RD) reads the required data from the row-buffer using the column multiplexer; and (WR) writes the data into the row-buffer using the column muxtiplexer. RD/WR commands are also called CAS. Additionally, a Refresh command is issued regularly to refresh DRAM capacitors.

The memory controller must satisfy different timing constraints that occur between various DRAM commands. The timing constraints are taken from the JEDEC standard [2] and are listed in Table 10.1 with values for DDR3-1333MHz device. We consider 1333 MHz as this speed is approximately in the middle of DDR3.

Parameters	Description	DDR3	Unit
t_{CK}	DRAM clock cycle	1.5	nsec
BL	Burst length of data bus	8	cols
CL	Read latency	9	cycles
WL	Write latency	7	cycles
t_{RCD}	ACT to Read/Write delay	9	cycles
t_{RP}	PRE to ACT delay	9	cycles
t_{WR}	Data end of Write to PRE delay	10	cycles
t_{WTR}	Write to Read delay	5	cycles
t_{RC}	ACT to ACT delay (same bank)	33	cycles
t_{RRD}	ACT to ACT delay (diff. bank)	4	cycles
t_{FAW}	Four ACT window	20	cycles
t_{RTP}	Read to PRE delay	5	cycles
t_{RAS}	ACT to PRE delay	24	cycles
t_{RTW}	Read to Write delay	7	cycles
t_{RFC}	Time to refresh a row	160	nsec
t_{REFI}	Average refresh interval	7.8	usec

Table 10.1: DRAM timing constraints [2].

Here we briefly mention different characteristics of the DRAM-system that influence its memory access time. The details can be found in [11].

1. *Row-buffer locality*. Since a row-hit requires fewer steps than a row-conflict, the latency of a row-hit is less than a row-conflict and this is called *row-buffer locality*.

2. *Bank-level conflicts* occur when multiple requests access the same bank. It results in a higher number of row-conflicts to the same bank, consequently the requests are serviced completely serially, and in this case, the latency is increased significantly.

3. *The direction of the data bus* should be changed upon the requests' sequence read-to-write or write-to-read and results in read-to-write latency and write-to-read latency respectively. During this time the data bus cannot be utilized. This latency exists whether the requests are made to the same bank or different banks.

4. *Scheduling algorithm* FR-FCFS, that unfairly prioritizes threads with high row-buffer locality.

Bank-level conflicts can be reduced by using *private banks* (supported by few hardware architectures like Freescale p4080 or by OS-based bank partitioning [12]). Other three characteristics are usually taken care in the memory-interference delay analysis for using private and/or interleaved banks [5, 6].

10.3 System model

We assume a single-chip multicore processor with a set of identical cores that have uniform access to the main-memory. Each core has a set of local resources (primarily a set of caches for instructions and/or data and busses to access these from the cores) and a set of resources that are shared amongst all cores (typically a Last-Level Cache (LLC), a main-memory, a memory bus, and the LLC and DRAM are connected by a command bus and a data bus). The architecture like Intel i5 3550, etc. complies with these assumptions. For simplicity, we consider one rank and a single channel. More than one channel can be considered independently since each channel has a separate command and data bus.

We assume that a local cache miss is stalling, which means whenever there is a miss in a LLC, the core is stalling until the cache-line is fetched from memory. We assume that all memory requests from the LLC to the shared DRAM go through the same channel and the data and command busses can be used in parallel. DRAM controller actually queues requests, however, at any given time only one of these requests is being served by the channel. Since the data is transferred in the *burst-mode* (a burst read/write allows to read/write the whole cache line after specifying only its start address, for a DIMM in a COTS-system has a cache line size of 64 B, a burst is 8 consecutive 64-bit pieces of memory), therefore, a single memory request can cache the data of one cache miss, thus the number of LLC misses is equal to the number of generated memory requests. Similar to [13, 6] we assume that each task has its own private partitions in the cache [14] that is sufficient to store one row of a DRAM bank. Further, we assume that cache-related preemption delays (CRPD) [15] are zero due to the partitioned cache.

We assume that the multicore processor uses DDR DRAM as its main memory, and it is not put on low power state at any time. The memory controller uses *open row policy* and employs FR-FCFS policy similar to [6] and we assume that DRAM records the arrival times of memory requests when they arrive at the controller. DRAM bank partitioning is considered to divide banks into partitions where memory request can access one bank in DRAM. We assume both *private banks* and *interleaved banks* (where memory request can access all banks in DRAM) are available and only one can be used at a time.

10.4 Memory interference delay analysis

We present an analysis of *worst case delay for a DRAM memory request ($D\ell$)*. The analysis depends on the hardware architecture and on the number of cores in the system. It is a sum of (1) worst case service time for a single memory request and (2) worst case delay this request under analysis can be delayed by other simultaneous requests (generated by other tasks executing on other cores).

10.4.1 Worst-case service time for a single memory request ($Dl_{ser.time}$)

We compute the worst-case service time for both, private banks (denoted as $Dl^p_{ser.time}$) and interleaved (or shared) banks ($Dl^s_{ser.time}$). We consider the worst cases of all previously mentioned characteristics that influence the memory access time of DRAM, i.e., row-conflicts, a change in the data bus direction for each request, and rescheduling algorithm. Since a row-conflict consists of three DRAM commands: ACT, PRE and CAS (RD/WR) (see Figure 10.1), thus $Dl_{ser.time}$ is a sum of latencies for ACT, PRE and CAS commands plus the DRAM timing constraints (given in Table 10.1) to meet these three commands.

$$\begin{aligned} Dl_{ser.time} &= (Dl_{PRE} + Dl_{ACT} + \max(TC_{PRE}, TC_{ACT}) + \\ &\quad Dl_{CAS} + TC_{CAS}) \times t_{CK} \end{aligned} \tag{10.1}$$

where $Dl_{PRE}, Dl_{ACT}, Dl_{CAS}$ present latencies for ACT, PRE and CAS commands respectively, while $TC_{PRE}, TC_{ACT}, TC_{CAS}$ present the timing constraints for these commands respectively.

$Dl^{p}_{ser.time}$ for private banks

The worst-case service time is denoted by $Dl^{p}_{ser.time}$, and all other latencies are also presented with ^{p} symbol.
PRE command latency: $Dl^{p}_{PRE} = t_{CK}$ each command takes one clock cycle on address/command bus.
ACT command latency: According to the JEDEC standard [2] (see Table 10.1), t_{RRD} is the minimum separation time between two ACT commands to different banks. And maximum four ACT commands can be issued during one t_{FAW} window. To consider the worst-case, we take the max of both as
$Dl^{p}_{ACT} = \max(t_{RRD}, t_{FAW} - 3.t_{RRD})$.
CAS command latency: CAS latency is the sum of RD/WR latency plus the time to transfer the data on the data bus. The read (RD) and write (WR) latencies are CL and WL respectively (see Table 10.1). The data is transferred in burst mode on both the rising and falling edges of the double data rate DDR bus, therefore, the time to transfer the data is $BL/2$. Thus the total time for RD command is $CL + BL/2$, and for WR command is $WL + BL/2$. For worst-case, we take the maximum of RD and WR latencies, i.e.,
$Dl^{p}_{CAS} = \max(CL + BL/2, WL + BL/2)$.
PRE, ACT and CAS commands' timing constraint latencies: For private banks, there is no timing constraint for PRE and ACT commands, thus $TC^{p}_{PRE} = 0$ and $TC^{p}_{ACT} = 0$.

The timing constraints for CAS commands are due to the change in the data flow direction of the data bus. It depends upon whether the direction of the data

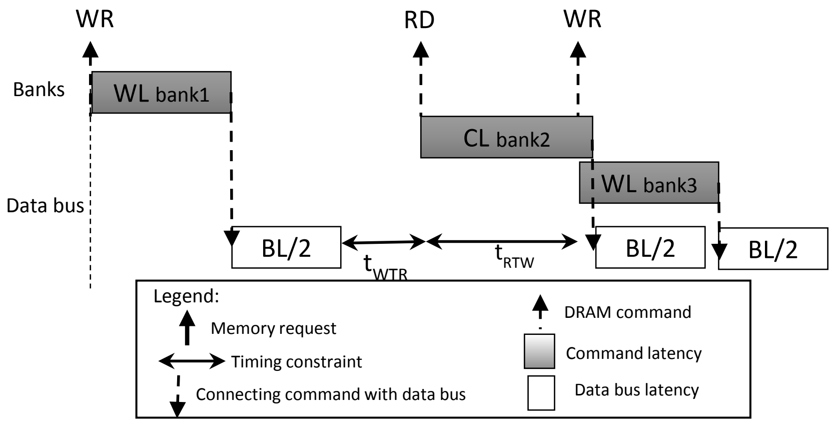


Figure 10.2: Timing constraints CAS commands for private banks (WR to bank1, RD to bank2, WR to bank3)

bus is switching from Write to Read, or from Read to Write. It is zero if there is no switching in the direction. These constraints are depicted in Figure 10.2. In figures, the values of commands are not drawn according to the scale. t_{WTR} starts after the data is transferred ($BL/2$), however, t_{RTW} starts at the start of the RD command.

$$TC^{p}_{CAS} = \begin{cases} t_{WTR} & \text{if switching from write} \\ t_{RTW} - (CL + BL/2) & \text{if switching from read} \\ 0 & \text{if not switching,} \end{cases} \tag{10.2}$$

Putting the values of all these latencies in equation 10.1 provides the service time of a memory request using private banks.

$Dl^{s}_{ser.time}$ for shared banks:

For shared banks, the worst-case service time is denoted by $Dl^{s}_{ser.time}$, and all latencies are presented with ^{s} symbol.
PRE command latency: When memory requests are accessing the same bank then $Dl^{s}_{PRE} = t_{RP}$ (see Table 10.1).
ACT command latency: is $Dl^{s}_{ACT} = t_{RCD}$ (see Table 10.1).
CAS command latency: RD and WR latencies for shared banks are the same as for private banks, i.e., CL and WL respectively. The time to transfer the data

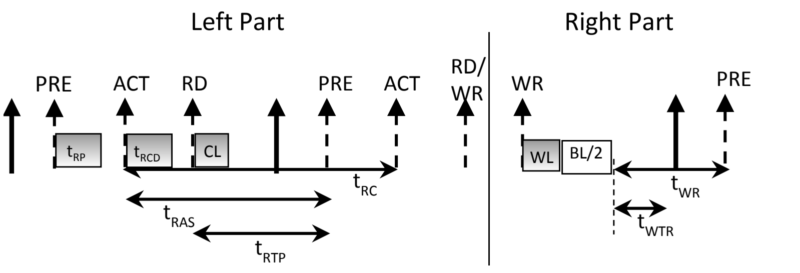


Figure 10.3: Timing constraints for shared banks

on the data bus is $BL/2$. Thus the total time for RD and WR is $CL+BL/2$, and $WL + BL/2$ respectively. Note that for RD command, data can be transferred in parallel (on the data bus) with the processing of data of the next command, therefore, $BL/2$ can be safely removed from all RD equations for simplicity. Thus RD latency becomes CL. For worst-case, we take the maximum of RD and WR latencies, i.e.,
$Dl^p_{CAS} = \max(CL, WL + BL/2)$.

PRE and ACT commands' timing constraint latencies: The timing constraints for PRE command depends on whether the previous command was RD or WR. It also depends on whether the row for the previous command was open or close.

Case 1: previous RD and open-row, (means only RD command was executed previously), thus t_{RTP} (RD to PRE delay) is considered (see Table 10.1). Since it includes the time to execute RD command (CL) (see left part of Figure 10.3), thus CL is subtracted from t_{RTP}. Thus the timing constraint is $\max(t_{RTP} - CL, 0)$.

Case 2: previous RD and close row, (means ACT and RD commands were executed for the previous request), so the timing constraint is taken from the ACT command of the previous request until the PRE command of the current request. t_{RAS} is the delay from ACT till PRE (see Table 10.1). It includes the time to execute previous ACT and RD commands within it (see left part of Figure 10.3). To calculate the timing constraint, the execution times of previous ACT and RD commands are subtracted from t_{RAS}; thus it becomes $t_{RAS} - T_{RCD} - CL$.

Case 3: previous WR and open-row, the timing constraint is $t_{WR} - t_{WTR}$ (see right part of Figure 10.3).

Case 4: previous WR and close-row is similar to case 2, only RD is replaced

by WR latency, i.e., $t_{RAS} - t_{RCD} - (WL + BL/2)$. Thus, for RD and WR commands, TC^s_{PRE} becomes
$TC^s_{PRE(RD)} = \max(t_{RTP} - CL, t_{RAS} - t_{RCD} - CL, 0)$.
$TC^s_{PRE(WR)} = \max(t_{WR} - t_{WTR}, t_{RAS} - t_{RCD} - (WL + BL/2))$.
For worst case we take the maximum of both, i.e.

$$TC^s_{PRE} = \max(TC^s_{PRE(RD)}, TC^s_{PRE(WR)}) \tag{10.3}$$

For timing constraint of ACT, t_{RC} is considered for the shared bank (see Table 10.1). It is the time starting from one ACT till the start of the next ACT to the same bank, therefore, it includes the delays of CAS and PRE commands within it. To compute the timing constraint of ACT, the latencies of ACT and CAS commands are subtracted from it (see left part of Figure 10.3).
$TC^s_{ACT} = t_{RC} - t_{RCD} - \min(CL, (WL + BL/2))$
TC^s_{ACT} includes the timing constraint for PRE command TC^s_{PRE} within it (see left part of Figure 10.3). Since we take maximum of TC_{PRE}, TC_{ACT} in equation 10.1, so in worst case, the TC^s_{ACT} will be chosen when row is closed. And if the row is open then ACT command will not be executed and TC^s_{ACT} is zero, thus TC^s_{PRE} would be chosen there. The analysis in [6] does not consider both these timing constraints (i.e. TC_{PRE} and TC_{ACT}) for shared banks. Timing constraints for the shared banks are higher than the private banks and a main source of increased latencies.
CAS command's timing constraint latency: depends upon the previous CAS command:

$$TC^s_{CAS} = \begin{cases} T_{WTR} & \text{if previous write} \\ 0 & \text{if previous read,} \end{cases} \tag{10.4}$$

10.4.2 Per-request interference delay

It is the interference delay to execute the number of memory requests present in the memory controller and to be served before the request under analysis. If M is number of cores, then $M - 1$ requests from other cores will be there in worst-case. Because of our assumption that core is stalling until the cache-line is fetched from memory, the maximum number of requests does not increase M. Considering the worst-case service time for each request (as presented in the previous Section 10.4.1), the interference delay becomes $(M-1) \times Dl_{ser.time}$. Adding the service time of the request under analysis, the total time to serve the request including interference becomes $Dl = Dl_{ser.time} + (M-1) \times Dl_{ser.time}$ or simply

$$Dl = M \times Dl_{ser.time} \tag{10.5}$$

For the private banks, $Dl_{ser.time}$ is substituted by $Dl^{p}_{ser.time}$ in the above equation. However for the shared banks, the reordering effect should also be taken into account.

Consecutive row-hit requests: According to FR-FCFS policy, the row-hit requests are given priority over the row-conflict requests. Row-hit requests are reordered at the bank scheduler and served before row-conflict requests. For worst case for m consecutive row-hit requests, we consider alternate read and write requests (means a change in the direction of data bus at each request). The worst-case service time for m consecutive row-hits is $Dl_{conhit}(m) = \{\lceil m/2 \rceil \times (WL+BL/2)+\lfloor m/2 \rfloor \times CL+m \times \max(case1, case3)+TC^{s}_{PRE}\}$. Since ACT command is not issued for open-rows, timing constraints for ACT are not included in $Dl_{conhit}(m)$. Also case1 and case3 (from last section) are included for open-rows only. TC^{s}_{PRE} of eq 10.3 is added if a PRE command is issued after m consecutive hit requests.

The maximum row-hits served by the system are N_{cols}/BL where N_{cols} is the number of columns in one row. In order to bound the reordering effect, a hardware threshold N_{cap} is also supported [10]. Thus in worst case the maximum number of row-hits prioritized over older row-conflicts is $N_{reorder} = \min(N_{cols}/BL,\ N_{cap})$ [6]. Substituting this number for m in Dl_{conhit}, i.e. $Dl_{conhit}(N_{reorder})$ equation gives the maximum number of row-hits served before older row-conflicts. $N_{reorder}$ can be greater than M. The assumption here is that each task can only have a single outstanding request, but once a hit from a task is served, it will unblock and can issue a new request that also results in a hit while the hits from other tasks are served. Considering worst-case, of $N_{reorder}$ hits and $M-2$ misses before the request under analysis, $Dl_{conhit}(N_{reorder})$ delay for hits and $Dl^{s}_{ser.time}$ for miss, the total delay becomes

$$\begin{aligned} Dl \quad = \quad & \max(Dl_{conhit}(N_{reorder}) + (M-2) \times Dl^{s}_{ser.time}, \\ & M - 1 \times Dl^{s}_{ser.time}) \end{aligned}$$

In case of no hits $N_{reorder} = 0$, and Dl becomes $M \times Dl^{s}_{ser.time}$.

According to [5], the refresh effect is added as $k^{i+1} = \left\lceil \frac{(Totalmemoryinterferencedelay+k^{i}) \times t_{RFC}}{t_{REFI}} \right\rceil$ and $k^{0} = 0$. Thus for

DDR3-1333H, t_{RFC}/t_{REFI} is $160ns/7.8\mu s = 0.02$, thus will increase the total memory interference delay by 2%.

10.5 Conclusions and future work

In this paper, we have safely bounded the memory contention for DDR DRAM memory controller that are commonly used in COTS multicore architectures. We have presented the worst case delay analysis of a memory request for private and shared memory banks. The analysis depends on the hardware architecture and on the number of cores. It is independent of the number of tasks executing in the system.

Previously, we have proposed a multi-resource server (MRS) [16, 17] approach to bound memory interference from other servers executing concurrently on other cores. The memory bandwidth has added as an additional server-resource to bound memory interference by considering a constant memory access time. In future, we intend to update the schedulability analysis of MRS by assuming a variable access time for the memory requests and combining our current analysis of $(D\ell)$ for this purpose.

Bibliography

[1] R. Rettberg and R. Thomas. Contention is no obstacle to shared-memory multiprocessing. *Commun. ACM*, 29(12):1202–1212, December 1986.

[2] Micron. 2Gb DDR3 SDRAM.

[3] S. Schliecker and M. Negrean and R. Ernst. Bounding the Shared Resource Load for the Performance Analysis of Multiprocessor Systems. In *Proc. of the Conference on Design, Automation and Test in Europe (DATE' 10)*, pages 759–764, 2010.

[4] R. Pellizzoni and A. Schranzhofer and J.-J.Chen and M. Caccamo and L. Thiele. Worst Case Delay Analysis for Memory Interference in Multicore Systems. In *Proc. of the Conference on Design, Automation and Test in Europe (DATE' 10)*, pages 759–764, 2010.

[5] Z-P. Wu, Y. Krish, and R. Pellizzoni. Worst case analysis of DRAM latency in multi-requestor systems. In *Proc. 34th IEEE Real-Time Systems Symposium (RTSS' 13)*, pages 372–383, December 2013.

[6] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. R. Rajkumar. Bounding memory interference delay in COTS-based multicore systems. In *Proc. 20th IEEE Real-Time Technology and Applications Symposium (RTAS' 14)*, pages 145–154, April 2014.

[7] J. Reineke, I. Liu, H.D. Patel, S. Kim, and E.A. Lee. PRET DRAM controller: Bank privatization for predictability and temporal isolation. In *(CODES+ISSS)*, September 2011.

[8] L. Yonghui, B. Akesson, and K. Goossens. Dynamic command scheduling for real-time memory controllers. In *Proc. of the 24th Euromicro Conf. on Real-Time Systems (ECRTS' 14)*, 2014.

[9] K.J. Nesbit, N. Aggarwal, J. Laudon, and J.E. Smith. Fair queuing memory systems. In *In International Symposium on Microarchitecture (MICRO)*, 2006.

[10] T. Moscibroda and O. Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *USENIX Security Symposium*, pages 257–274, 2007.

[11] R. Inam and M. Sjödin. Combating unpredictability in multicores through the Multi-Resource Server. In *Workshop on Virtualization for Real-Time Embedded Systems (VtRES'14)*. IEEE, September 2014.

[12] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *Proc. of the 21st International Conf. on Parallel Architectures and Compilation Techniques (PACT' 12)*, pages 367–376, 2012.

[13] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory- access Control in Multiprocessor for Real-Time Systems with Mixed Criticality. In *Proc. of the 24th Euromicro Conf. on Real-Time Systems (ECRTS' 12)*, July 2012.

[14] J. Liedtke, H. Härtig, and M. Hohmuth. OS-controlled cache predictability for real-time systems. In *Proc. 3rd IEEE Real-Time Technology and Applications Symposium (RTAS' 97)*, pages 213–224, June 1997.

[15] D. Hardy and I. Puaut. Estimation of cache related migration delays for multi-core processors with shared instruction caches. In *17th International Conference on Real-Time Networks and Systems (RTNS' 09)*, October 2009.

[16] M. Behnam, R. Inam, T. Nolte, and M. Sjödin. Multi-core composability in the face of memory bus contention. *ACM SIGBED Review, Special Issue on 5th Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, 10(3), 2013.

[17] R. Inam, N. Mahmud, M. Behnam, T. Nolte, and M. Sjödin. The Multi-Resource Server for predictable execution on multi-core platforms. In *Proc. 20th IEEE Real-Time Technology and Applications Symposium (RTAS' 14)*, pages 1 – 11, April 2014.

Chapter 11

Paper F: Compositional analysis for the Multi-Resource Server – a technical report

Rafia Inam, Moris Behnam, Thomas Nolte, Mikael Sjödin
MRTC report ISSN 1404-3041, ISRN MDH-MRTC-283/2014-1-SE, Mälardalen Real-Time Research Center, Mälardalen University, October, 2014 (submitted to conference).

Abstract

The Multi-Resource Server (MRS) technique has been proposed to enable predictable execution of memory intensive real-time applications on COTS multicore platforms. It uses resource reservation approaches in the context of CPU-bandwidth and memory-bus bandwidth reservations to bound the interferences between the applications running on the same core as well as between the applications running on different cores.

In this paper we present a complete compositional schedulability analysis for the Multi-Resource Server technique. We present how memory contention for contemporary DDR DRAM memory-architectures be can safely bounded for schedulability analysis in the context of the Multi-Resource Server. Based on the proposed analysis, we further provide an experimental study that investigates the behaviour of the MRS and identify the factors that contribute mostly on the overall system performance.

11.1 Introduction

With the advent of highly efficient multicore architectures, multiple real-time applications are integrated together and are executed concurrently on multicore platforms. As a result, these applications share not only CPU with each other, but also other physical resources of the multicore architecture (like shared caches, memory-bus bandwidth, and memory). Contention for the shared physical resources is a natural consequence of sharing [1]. It does not only reduce throughput but also affects the predictability of real-time applications.

On unicore platforms, the server-based scheduling has been developed to achieve predictable integration by successfully bounding the interference between integrated applications [2, 3, 4]. However, this approach is CPU centric and is limited in managing the CPU-resource only. It does not account for activities that are located on different cores and thus still allow interference amongst applications in an unpredictable manner. For multicore platforms, a solution has proposed to solve these problems through updating the traditional server-based scheduling approach with a novel memory aware Multi-Resource Server (MRS) technology [5, 6] for Commercial Off-The-Shelf (COTS) multicore hardware.

MRS enables predictable execution of real-time applications on multicore platforms through resource reservation approaches in the context of CPU-band width reservation and memory-bus bandwidth reservation. The MRS provides temporal isolation, both between tasks running on the same core (through CPU partitioning), as well as between tasks running on different cores (through memory-bus bandwidth partitioning). The latter could, without MRS, cause interfere due to contention on the shared memory bus. A local analysis for tasks executing in an MRS considering a constant memory access time has been presented in [5].

Contributions: In this paper we update the local analysis by relaxing this assumption and considering the worst case delay for accessing memory requests in our analysis. We present a complete and composable global schedulability analysis for both resources (CPU- and memory-bandwidth) of the MRS. Further, we provide a study that brings insight on how these both resources relate to each other. In addition, the evaluation shows the effect of changing the priority of a memory-intensive task on both of these resources.

The preliminary work [5] focused on just the presentation of the general idea of the MRS and described its initial local schedulability analysis. It did not address the global schedulability analysis and lacked an investigation study. In this paper we complete the local and the global analysis and perform an experimental study to investigate the behavior of the MRS and the factors effecting its behavior.

Paper Outline: Section 11.2 presents the related work on server-based and memory access techniques for multicore systems. Section 11.3 explains our system model. The local and global schedulability analysis is described in Section 11.4. The correlation between (1) CPU and memory budgets, (2) private and shared memory banks, and the impacts of (1) period of memory-intensive task and (2) period of the MRS on server-budgets is investigated in Section 11.5, and finally Section 11.6 concludes the paper.

11.2 Related work

The problem of contention of shared resources has gained a significant importance in the context of multicore embedded systems. Hierarchical scheduling is one technique to provide predictable execution on unicore platforms [4, 7, 8]. Solutions for multi-core architectures are based on strong (often unrealistic) assumptions on no interference originating from the shared hardware [9]. For multicore architectures, the assumption no longer remains valid.

Some highly predictable Time Division Multiple Access (TDMA) based techniques are used for memory bus arbitration. Rosen et al. [10] measured the effects of cache misses on the shared bus traffic where the memory accesses are confined at the beginning and at the end of the tasks. Schranzhofer et al. [11] relaxed this assumption of fixed positions. They divide tasks into sets of superblocks that are specified with a maximum execution time and a maximum number of memory accesses. Another work that guaranteed a minimum bandwidth and a maximum latency for each memory access was proposed by Akesson et al. [12] using a two-step approach to share a predictable SDRAM controller. These techniques eliminate the interference of other tasks through isolation; however, they are limited in the usage of only a specified (non-COTS) hardware.

Schliecker et al. [13] have bounded the shared resource load by computing the maximum number of consecutive cache misses generated during a specified time interval. The joint bound is presented for a set of tasks executing on the same core covering the effects of both intrinsic and pre-emption related cache misses. A tighter upper bound on the number of requests is presented by Dasari et al. [14] by using non-preemptive task scheduling. However, these works lack the consideration of independently developed subsystems and the use of memory servers to limit the access to memory bandwidth. The main focus of our work is on both aspects (compositionality and independent development) w.r.t. server-based methods. Pellizzoni et al. [15] proposed the division of tasks into superblock sets by managing most of the memory request either at the start or at the end of the execution blocks. This idea of superblocks was later used in TDMA arbitration [11]. All these techniques assume a constant access time for each memory request and do not consider the reordering of requests.

Recent works [16, 17] considered variable access time of memory requests for tasks executing concurrently on different cores and contending for memory accesses. [16] considered private banks for each requestor, using a FIFO ordering for serving requests, by considering one queue for each bank and a global queue to accumulate requests from each bank. The work in [17] provided analysis for both private and shared DRAM banks and considered the First Ready First Come First Serve (FR-FCFS) scheduling policy to account the reordering effect. However, these works considered the task-level schedulability only and lack the consideration of independently developed subsystems and the use of memory servers to limit the access to memory bandwidth, which is our main focus.

A server-based approach to bound the memory load of low priority non-critical tasks executing on non-critical cores was presented in [18]. Memory servers are used to limit memory requests generated by tasks of non-critical cores. A response time analysis is proposed for tasks that are located on critical cores, including the interference that can be generated from non-critical cores, considering a constant access time for memory requests. We propose a more general approach to support both composability and independent development of subsystems by using servers on all cores. The analysis in [18] only considers one memory server on each non-critical core while we present analysis for both time and memory aspects of the servers executing on all cores and consider multiple servers per core. An analysis for variable DRAM access time to serve memory requests is presented in [19], and it is used in the schedulability analysis of our MRS in Section 11.4.

11.3 System model

Here we present our hardware platform, the system model and the assumptions we follow.

11.3.1 Architecture

We assume a single-chip multicore processor with a set of identical cores. Each core has a set of local resources; primarily a set of caches for instructions and/or data and busses to access these from the cores. The system has a set of resources that are shared amongst all cores: this is typically a Last-Level Cache (LLC), a main-memory (DRAM) and a shared memory bus. The architectures like Intel core 2 CPU 6700, Intel i5 3550, etc. comply to these assumptions.

In this work we assume that a local cache miss is stalling, which means whenever there is a miss in a LLC, the core is stalling until the cache-line is fetched from memory. We assume that all memory requests from the LLC to the shared DRAM go through the same bus, and that the bus serves one request at a time.

We assume that the multicore processor uses Double Data Rate Dynamic RAM (DDR DRAM) as their main memory resource [20], which is shared amongst all of the cores. The controller employs *First-Ready First Come First Served (FR-FCFS)* scheduling policy [21], that prioritizes the ready DRAM commands (row-hit memory requests) over others and for ties, it prioritizes older requests in order to improve row-hit ratio and maximize the overall throughput. DRAM bank partitioning (or *private banks*) is considered to divide the banks into partitions where memory request can access one bank in DRAM. Many COTS architectures do not support private banks, but it can be achieved through operating system bank partitioning [22]. We assume both *private banks* and *interleaved or shared banks* (where memory request can access all banks in DRAM) are available and only one type can be used at a time similar to [17]. The DRAM model is used to compute the worst case delay for a DRAM memory request. More details on DRAM background, DRAM model, and memory interference delay analysis can be found in [19].

11.3.2 Server model

Our scheduling model for the multicore platform can be viewed as a set of trees, with one parent node and many leaf nodes per core, as illustrated in Figure 11.1. The parent node is a *node scheduler* and leaf nodes are the subsystems (servers). Each subsystem contains its own internal set of tasks that are scheduled by a *local scheduler*. The node scheduler is responsible for dispatching the servers according to their bandwidth reservations (both CPU- and memory-bandwidth). The local scheduler then schedules its task set according to a server-internal scheduling policy.

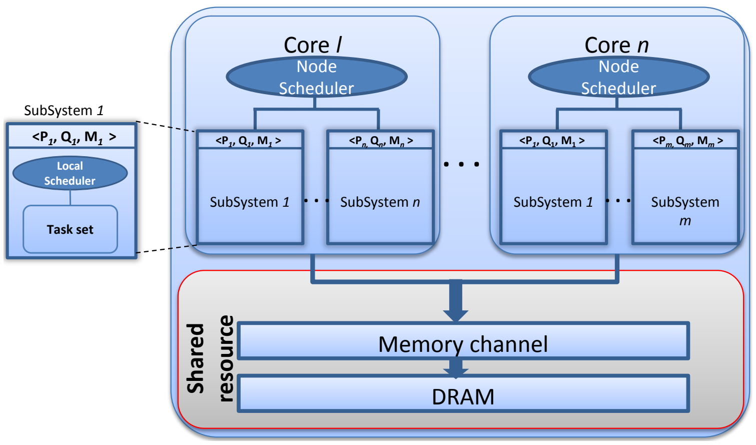


Figure 11.1: A multi-resource server model

We follow the same model for the MRS. Each MRS S_s is allocated a period and two different budgets, according to $\langle P_s, Q_s, M_s \rangle$, where P_s is the period of the server, a CPU budget Q_s is the amount of CPU-time allocated each period, and a memory-bandwidth budget M_s is the number of memory requests in each period. The CPU-bandwidth of a server is thus Q_s/P_s and we assume that the total CPU-bandwidth is not more than 100%.

During run-time, each MRS is associated with two dynamic attributes q_s and m_s which represent the amount of available CPU- and memory-budgets respectively. For both levels of schedulers, including the node and server-level, the *Fixed Priority Pre-emptive Scheduling (FPPS)* policy is implemented.

We assume that each server is assigned to one core and that its associated tasks will always execute only on that core, i.e., task or server migration is not allowed.

The MRS is of periodic type, i.e., it replenishes both CPU- and memory-budgets to the maximum values periodically. At the beginning of each server period its dynamic attributes are set as $q_s := Q_s, m_s := M_s$. In each core, the node scheduler is responsible to schedule all ready servers and it selects a highest priority ready server for execution. A server is ready to execute if it possess *both* remaining CPU- and memory-budgets, formally: $(q_s > 0 \wedge m_s > 0)$. A higher priority server can pre-empt the execution of lower priority servers. During the server's execution, its CPU-capacity, q_s, decreases with the progression of time, while its memory-bandwidth capacity, m_s, decreases when a task in the server issues a memory request. A server which depletes any of its resources is suspended from execution and waits for its replenishment at the beginning of the next server-period. Thus, if any of the budgets is depleted then the other remaining budget will be discarded, i.e., if $m_s = 0$ or $q_s = 0$ then $m_s = q_s = 0$

The idling periodic server strategy [23] is used for CPU reservation, i.e., if the scheduled server has remaining budget but there is no task ready then it simply idles away its CPU-budget until a task becomes ready or one of the server's budgets depletes. A scheduled server uses its local scheduler to select a task to be executed. A higher priority task can pre-empt the execution of lower priority tasks but not during the core is stalling. The details of the implementation of MRS and its execution can be found in [6].

11.3.3 Task model

We are considering a simple sporadic task model in which each task τ_i is represented as $\tau_i(T_i, C_i, D_i, CM_i)$ where T_i denotes the minimum inter-arrival time of task τ_i with worst-case execution time C_i and deadline D_i where $D_i \leq T_i$. The tasks are indexed in reverse priority order, i.e. τ_i has priority higher than that of τ_{i+1}.

CM_i denotes the maximum number of cache miss requests and the time of issuing a cache miss request is arbitrary during the task's execution time. Similar to [18, 17] we assume that each task τ_i has its own private partitions in the cache that is sufficient to store one row of a DRAM bank. This assumption can be satisfied by implementing operating system based cache coloring [24]. Further, we assume that cache-related preemption delays (CRPD) [25, 26] are zero due to partitioned cache, and the value of CM_i does not change due to preemption.

11.4 Schedulability analysis

We use the compositional hierarchical schedulability analysis techniques to check the system schedulability by composing the subsystems interfaces which abstract the resource demands of the subsystems [4]. The analysis is performed in two levels; the first is called the local schedulability analysis where for each subsystem its interface parameters are validated locally based on the resource demand of its local tasks. The second level is called the integration or the global schedulability level, where the subsystems interfaces are used to validate the composability of the subsystems.

11.4.1 Local schedulability analysis

First, we present the local schedulability analysis without considering the effect of the memory bandwidth part of the multi-resource server, i.e., assuming a simple periodic server, and then we extend the analysis to include the effect of the memory requests. Note that, at this level, the analysis is independent of the type of the server as long as the server follows the periodic model, i.e., both budgets are guaranteed every server period. We assume that the server's period, CPU-budget, and memory-budget are all given for each server.

Considering only CPU-budget

The local schedulability analysis under FPPS is given by [4]:

$$\forall \tau_i \ \exists t : 0 < t \leq D_i, \ \ \mathtt{rbf}_s(i,t) \leq \mathtt{sbf}_s(t), \tag{11.1}$$

where $\mathtt{sbf}_s(t)$ is the *supply bound function* that computes the minimum possible CPU supply to S_s for every time interval length t, and $\mathtt{rbf}_s(i,t)$ 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. $\mathtt{sbf}_s(t)$ is based on the periodic resource model presented in [4] and is calculated as follows:

$$sbf_s(t) = \begin{cases} t - (k-1)(P_s - Q_s) - BD_s & \text{if } t \in W^{(k)} \\ (k-1)Q_s & \text{otherwise,} \end{cases} \tag{11.2}$$

where $k = \max\Big(\big\lceil\big(t + (P_s - Q_s) - BD_s\big)/P\big\rceil, 1\Big)$ and $W^{(k)}$ denotes an interval $[(k-1)P_s + \mathsf{BD}, (k-1)P_s + \mathsf{BD}_s + Q_s]$. *Blackout Duration* BD is the longest time interval that the server cannot provide any CPU resource to its internal tasks and it is computed as $\mathsf{BD}_s = 2(P_s - Q_s)$. The computation of BD guarantees a minimum CPU supply, in which the worst-case budget provision is considered, assuming that tasks are released at the same time when the subsystem budget has depleted, the budget has been served at the beginning of the server period, and the following budget is supplied at the end of the server period due to interference from other higher priority servers.

For the request bound function $\texttt{rbf}_s(i,t)$ of a task τ_i, to compute the maximum execution requests up to time t, we assume that τ_i and all its higher priority tasks are simultaneously released. $\texttt{rbf}_s(i,t)$ is calculated as follows:

$$rbf_s(i,t) = C_i + \sum_{\tau_k \in \texttt{HP}(i)} \left\lceil \frac{t}{T_k} \right\rceil \times C_k, \tag{11.3}$$

where $\texttt{HP}(i)$ is a set of tasks with priority higher than that of τ_i. Looking at (11.3), it is clear that $\texttt{rbf}_s(i,t)$ is a discrete step function that changes its value at certain time points ($t = a \times T_k$ where a is an integer number). Then for (11.1), t can be selected from a finite set of scheduling points $\{SP_i\}$.

Worst case delay for a DRAM memory request

$D\ell$ presents *worst case delay for a DRAM memory request.* The $D\ell$ analysis depends on the hardware architecture and on the number of cores in the system, private or shared banks, along with the scheduling policy of memory controller used to serve parallel requests. It is independent of the maximum number of requests that can be generated in all the other servers. $D\ell$ is a sum of (1) worst case service time for a single memory request and (2) worst case delay this request can be delayed by other simultaneous requests (generated by other tasks executing on other cores) served by the memory controller. $D\ell$ is computed for worst cases for both private (denoted as $D\ell^p$) and shared DRAM banks (denoted as $D\ell^s$). The detailed memory interference delay analysis can be found in [19]. Since only one either private or shared can be used at a time, we only use the term $D\ell$ in the analysis. For experiments, we compute values for both $D\ell^p$ and $D\ell^s$, and use them accordingly.

Considering CPU- and memory-budget

In [13, 14, 18], the effect of the memory bandwidth access has been included in the calculations of the response times of tasks. The basic idea in all these works is the computation of the maximum interference $MI(t)$ caused by the memory-bandwidth contention on tasks during time interval t. The new request bound function including the memory-bandwidth contention is provided in (11.4).

$$rbf_s^*(i,t) = C_i + \sum_{\tau_k \in \texttt{HP}(i)} \left\lceil \frac{t}{T_k} \right\rceil \times C_k + MI(t). \tag{11.4}$$

In their approaches, $MI(t)$ is computed by multiplying the time needed for a request to be completed by the upper bound of memory-bandwidth requests in t, issued by all the other tasks (executed by all the other servers in [18]) located on cores other than the one hosting the analyzed task. However, this method cannot be used in our case since we assume that the subsystems are developed independently hence the tasks' parameters that belong to the other cores are not known in advance. In addition, the effect of both budgets (CPU and memory) should be accounted for in the MRS, which has not been considered in the previous works.

To solve this problem, we focus on the memory-bandwidth requests that can be generated by the tasks running inside a MRS. Considering the behaviour of MRS, we can distinguish two cases that can affect its tasks' execution.

1. When a task τ_i, executing in S_s, issues a memory request that causes a miss in a local cache, the associated core is stalling until the cache-line is fetched from memory. The maximum time that the task can be delayed due to the core in stalling state is presented as $D\ell$ and this delay should be considered in the analysis.

2. CPU-budget depletion due to memory-budget depletion. When tasks belonging to the same server issue M_s memory requests, the memory-budget will deplete which will force the CPU-budget to be depleted as well. In the worst case, M_s memory requests can be issued from tasks of the same server S_s sequentially, i.e. tasks send a new request directly after serving the current one. If this happens at the beginning of the server execution, a complete CPU-budget will be dropped and the server's internal tasks will not be able to execute during this server period (this case is shown in the first server period in Figure 11.2).

In the schedulability analysis of τ_i, we model the task delay due to core stalling as an interference from a fictive higher priority task $\tau*_{fic}$ with an execution-time equal to $D\ell$. The number of times that $\tau*_{fic}$ interferes with the execution of τ_i is defined as $NR(i,t)$ which equals to the *maximum number of memory requests* that can be generated at a time interval t. Note that during the execution of τ_i, it can be delayed by at most one memory request sent from a lower priority task and by the number of requests that the task itself sends and finally by the higher priority tasks that can preempt its execution. (11.5) is used to find $NR(i,t)$. Note that we include the delay due to a memory request sent from a lower priority task in the equation by adding one in $CM_i + 1$. Finally, $MI(t)$ in (11.4) is then calculated using (11.5)

$$NR(i,t) = CM_i + 1 + \sum_{\tau_k \in \mathtt{HP}(i)} \left\lceil \frac{t}{T_k} \right\rceil \times CM_k, \qquad (11.5)$$

$$MI(t) = NR(i,t) \times D\ell \qquad (11.6)$$

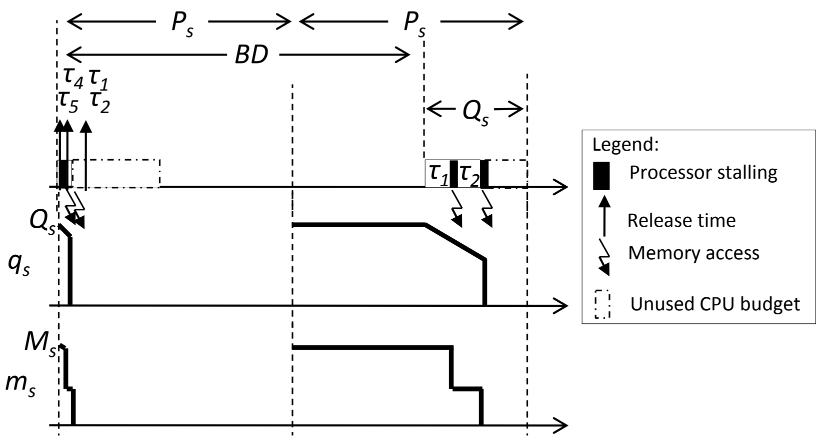


Figure 11.2: An example illustrating the worst-case CPU-supply

The second case that should be considered in the analysis is when the server CPU-budget depletes after sending M_s requests. This can happen when a task issues a memory-bandwidth request and then directly gets preempted after serving the request by a higher priority task that also issues a memory-bandwidth request and gets preempted by a third higher priority task and so on. This case affects the CPU resource supply that can be provided to the tasks. The basic assumption for computing $\mathtt{sbf}_s(t)$ is that the tasks are released when the CPU budget has been fully consumed and the budget was served at the beginning of the server period. However, as explained in the second case, the CPU-budget can deplete at the beginning of the server period (after serving M_s memory requests) because of the depletion of memory bandwidth budget. At any time t and for any task τ_i the maximum number of server periods that the server budget depletes due to the memory budget depletion, can be computed using the following function.

$$A(i,t) = \min\Big(\left\lfloor \frac{NR(i,t)}{M_s} \right\rfloor, \left\lceil \frac{t}{P_s} \right\rceil - 1 \Big) \qquad (11.7)$$

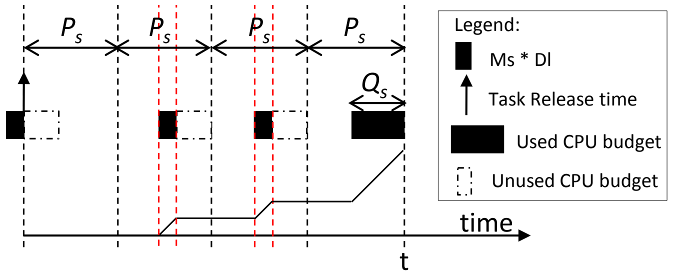


Figure 11.3: New supply bound function sbf*(i,t)

Note that the first part of the max function in (11.7) provides the upper bound of server periods that the budget can deplete due to the memory depletion during the time interval t, however, it cannot exceed the number of server periods up to t which is bounded in the second part of the max function.

By computing $A(i,t)$, we can consider the effect of the memory budget part on the $\texttt{sbf}_s(t)$ by assuming that $A(i,t)$ CPU budgets will not be provided up to t, i.e., the server budget $Q_s = 0$ whenever memory budget depletes. This can be achieved by increasing BD in (11.2) by $A(i,t) \times P_s$ which is equivalent to removing $A(i,t)$ CPU budgets, i.e., $A(i,t) \times Q_s$ from the supply bound function. However, the CPU budget will be depleted due to the depletion of memory budget only after serving M_s requests which is equivalent to providing $M_s \times D\ell$ CPU resource every server period as shown in Figure 11.3 (remember that each memory request delay $D\ell$ is modeled as an extra CPU demand in the $rbf_s^*(i,t)$). To decrease the pessimism in the analysis then we assume that $A(i,t) \times M_s \times D\ell$ will be added in the calculation of $\texttt{sbf}_s(t)$ which makes this function different for different tasks. However, it will be correct only if $M_s \times D\ell \leq Q_s$. The supply bound function $sbf_s^*(i,t)$ for τ_i is computed as follows.

$$sbf_s^*(i,t) = \begin{cases} t - (k(i,t)-1)(P_s - Q_s) - \\ BD_s(i,t) + A(i,t) \times M_s \times D\ell & \text{if } t \in W^{(k)}(i,t) \\ (k(i,t)-1)Q_s \\ +A(i,t) \times M_s \times D\ell & \text{otherwise,} \end{cases} \tag{11.8}$$

where $k(i,t) = \max\Big(\big\lceil\big(t + (P_s - Q_s) - BD_s(i,t)\big)/P\big\rceil, 1\Big)$ and $W^{(k)}(i,t)$ denotes an interval:

$[(k(i,t)-1)P_s + BD_s(i,t), (k(i,t)-1)P_s + BD_s(i,t) + Q_s]$
and $BD_s(i,t) = 2P_s - Q_s + A(i,t) \times P_s$.

11.4.2 System integration

During the integration phase of MRSes, all servers should be guaranteed to receive the required CPU and memory budgets specified in their interfaces. To validate this, two different tests should be applied. The first test is performed on the CPU part to make sure that the required CPU budget will be provided. The second test is performed to make sure that the total memory bandwidth usage by all servers in the system is lower than the maximum available bandwidth of the memory bus. Both tests can be performed independently and should succeed to guarantee that all tasks meet their deadlines. Therefore the parameters that are provided in the interface of each subsystem S_s to apply both tests are $P_s, Q_s, M_s, D\ell$.

As described earlier, the value of $D\ell$ depends on the hardware architecture. This keeps our local analysis independent of other servers in the system. As a simple example and assuming the FR-FCFS policy and knowing that only one request can be sent from each core at a time (since a core is stalling when a request is sent), then the upper bound value of $D\ell$ equals to the number of cores multiplied by the time taken to serve each request, plus adding the reordering effect of FR-FCFS policy, as presented in [19]. The reason is that for each core when it tries to send a memory request, as a worst case all other cores send one request just before the core under analysis, and one request from a lower priority task on executing on the same core, which bounds the number of requests.

Global schedulability test for CPU-budget: Since the CPU part of the server is of periodic type, each subsystem can be modeled as a simple periodic task where the subsystem period is equivalent to the task period and the subsystem budget is equivalent to the worst case execution time of a task. Then the schedulability analysis used for simple periodic tasks can be applied on all servers that share the same core for this test [4]. $R_i^{k+1} = Q_i + B_i + \sum_{S_j \in \mathtt{HEP}(i)} \left\lceil \frac{R_i^k}{P_j} \right\rceil \times Q_j$. The test is stopped when $R_i^{k+1} = R_i^k$ and $R_i^{k+1} \leq P_i$. Note that since for each memory request, the associated core is stalling then a higher priority server may be blocked by a lower priority server at most once with maximum blocking time equal to $B_i = D\ell$. This blocking time is considered in the analysis.

Global schedulability test for memory-budget: The total sum of request rates among all servers on all cores (B^{max}) should be less than the minimum bandwidth (or service rate) of the memory bus. This is similar to the requirement of CPU bandwidth reservation in that the total sum of CPU bandwidth reservation must be equal or less than 100% of CPU bandwidth. The practical minimum service rate (B^{avail}) that can be used to access data from DRAM has some practical limit and is less than the maximum bandwidth of the bus. It is difficult to obtain this bound from documentation, therefore, it is experimentally measured. The practical minimum service rate measured for Intel Core2Quad Q8400 processor is $B^{avail} = 1198MB/s$ [27]. We experimentally measured it $B^{avail} = 1022MB/s$ for our Intel core 2 CPU 6700 architecture.

For global schedulability test, that max bandwidth (B^{max}) used by all servers on all cores should not exceed this limit $B^{max} \leq B^{avail}$. As the memory-bus is shared among all cores, therefore, we sum up all requests from all servers from all cores. B^{max} is computed as $B^{max} = \sum_{\forall S_i} \left(\frac{M_i}{P_i} \times 64 \times 1000/(1024 \times 1024) \right)$. The number of memory requests are converted to the bandwidth by dividing with server period P_i, multiplying it with the size of cache line (i.e. 64bits). To convert the service rate to MB/s, it is multiplied with 1000 and divided by (1024×1024).

11.5 Investigating CPU- and memory-budgets

In this section we investigate the relationship of CPU- and memory-budgets and the effect of increase/decrease of memory-budget M_s on CPU-budget Q_s of a MRS S_s using synthetic experiments. We look into the effects of private and shared memory banks on the server.

11.5.1 Evaluation setup

We consider a multicore system using quad-processors, and DDR3-DRAM 1333H memory controller with 8 banks per rank. COTS architectures with these specifications are available (e.g. Intel Core i5). The upper bound of $D\ell$ is computed for both private ($D\ell^p$) and shared banks ($D\ell^s$) in nano seconds (ns) and the value of $D\ell^s$ is almost double than that of $D\ell^p$.

Two different task behaviours

Two different synthetic task-types are used in our synthetic evaluations, namely *normal task*, and *memory intensive task*. The normal task generates a relatively low number of memory requests ($CM_i = 1000$) per task period as compared to the memory intensive task. The memory intensive task generates higher number of memory requests ($CM_i = 20000$) per task period. Thus this task will heavily affect the memory budget requirements of the server and will also effect the CPU-budget of the associated MRS indirectly.

Timing properties of a MRS and its task set

Since we have previously shown the composition of MRSes in [6], in this paper we focus on the individual behaviour of a single server and how a server's parameters are affected from its tasks.

A single MRS is considered for the experiments with a period of 60 ms, and consisting of three tasks: two normal tasks and one memory-intensive task. A normal tasks generates 1000 memory requests per task period, while a memory-intensive task generates 20000 requests per task period. The timing properties of the three tasks are presented in Table 11.1.

Tasks	τ_1	τ_2	τ_3
Priority	H	M	L
Period (ms)	160	320	640
WCET (ms)	3	4	9
CM	1000	1000	20000

Table 11.1: Task properties.

Calculating minimum and maximum memory-budgets

We assume that the server period is given similar to [4], which is required to evaluate both CPU and memory budgets. We first calculate a range of minimum and maximum values for the memory-budget, and then for each value within the range, we evaluate the minimum CPU-budget so that the system remains schedulable using equations 11.1, 11.4, 11.8. The minimum and maximum values (M_{min} and M_{max}) represent minimum and maximum bounds for the memory-budget of the server respectively. From the memory perspective, each task should be able to issue all its memory requests CM_i within its period T_i and its server should serve these requests within T_i, otherwise, the task will miss its deadline. Thus $M_{min} = \max \forall_{\tau_i}(CM_i / \left\lfloor \frac{T_i - P_s}{P_s} \right\rfloor)$ and $M_s \geq M_{min}$ condition should be satisfied. M_{max} is computed as $M_{max} = \max(\forall_{\tau_i} NR(i, T_i))$. $NR(i, T_i)$ represent the maximum number of generated requests till the deadline of task τ_i, and for M_{max} we consider that all these requests are generated in one server period M_s. More than these requests cannot be generated during this period.

11.5.2 Synthetic experiments

The main focus of performing synthetic experiments is to investigate the behaviour of MRS by changing different parameters, like: (1) changing the value of memory-budget and exploring its affect on the CPU-budget's value; (2) checking the effect of private and shared memory banks; (3) changing the priority of memory-intensive tasks and investigating its effect on both budgets; and (4) increasing memory request of a task and observing its effect on both budgets of the server. M_{min} and M_{max} values are computed and the experiments are conducted for that range of M_s. The upper bound values for $D\ell^p$ and $D\ell^s$ are used for private and shared memory banks respectively. The analysis to compute $D\ell^p$ and $D\ell^s$ is presented in [19].

Experiment 1: Correlation between CPU- and memory-budgets

The purpose of this experiment is to investigate the correlation between both budgets M_s and Q_s. The timing properties of the MRS and its tasks set used for this experiment are presented in Table 11.1. The results are presented in Figure 11.4 where the x-axis denotes the range of M_s, and the y-axis shows the minimum CPU-budget Q_s for which the server is schedulable. Note that for better presentation of the graph, we shortened the shown range of M_s values.

This graph shows a stair-function, and the value of Q_s decreases at certain points with the increase of M_s. The reason is that when the value of M_s is minimum, and memory requests are generated at the start of the server period, then M_s depletes after $M_s \times D_\ell$ time and all the remaining CPU-budget for that period is simply discarded (as shown in second and third server period in Figure 11.3). The demand of Q_s is high as more server periods are needed to execute the tasks. Conversely, the increase of M_s decreases the value of $A(i, t)$ in equation 11.7. This results in a decrease in $BD_s(i, t)$ (see equation 11.8), and an increase in $sbf_s^*(i, t)$, thus requiring less Q_s.

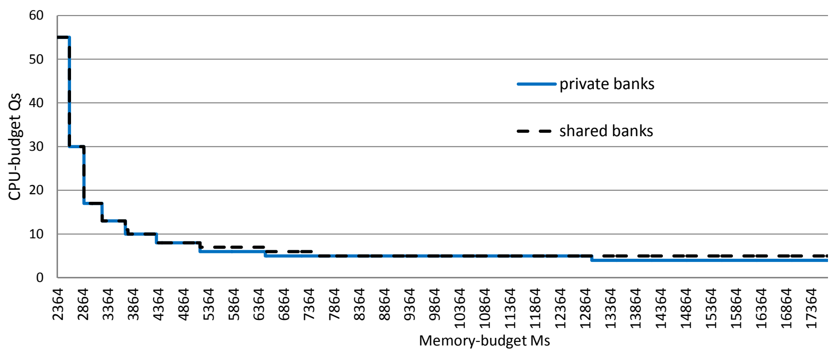


Figure 11.4: Correlation between M_s and Q_s considering private and shared memory banks.

The Effect of private and shared memory banks

Figure 11.4 presents the results of using both private and shared memory banks using $D\ell^p$ and $D\ell^s$ respectively. We note in this experiment that the choice of private or shared banks does not affect the needed CPU-allocation, Q_s, much. Often the needed allocation is the same regardless which memory organization is used. And in the rather few cases when private banks allow a smaller allocation of Q_s, the decrease in Q_s is negligible. It is mainly due to a big difference between time unit of memory-interference delay (nano sec) and the time unit of server period and CPU-budget (ms).

Experiment 2: Impact of the period of memory-intensive task on server budgets

We performed this experiment by changing the number of memory requests of the tasks in the previous experiment, i.e., first the high priority task τ_1 in Table 11.1 generates $20K$ requests, τ_2 and τ_3 generate 1000 requests each. It means that the period of memory-intensive task is 160. Second, the medium priority task τ_2 generates $20K$ (i.e. the period of memory-intensive task is 320) and τ_1 and τ_3 generate 1000 requests each. Third, the low priority task τ_3 generates $20K$ other tasks generate 1000 requests (i.e. the period of memory-intensive task is 640 now). Other properties of tasks remain the same as presented in Table 11.1.

We see in Figure 11.5, the need for memory-budget increases a lot when the memory-intensive task is executed with a higher priority. There are two reasons for this effect. First, as rate monotonic is used for priority assignment, therefore, the shorter period task activates more often than other longer period tasks, that leads to an increase in the total number of generated requests during a time interval (t). It increases the memory-budget M_s of the server period (obvious from the graph of $\tau 1 = 20K$ in Figure 11.5 where MRS is schedulable for a higher M_s value). The increase in the number of requests increases the value of $A(i,t)$ in (11.7), consequently $BD_s(i,t)$ increases as well (see eq. 11.8). Second, the higher priority task affects the request bound function of all the lower priority tasks by adding the memory interference delay $MI(t)$ to their rbf_s (see eq. 11.4), thus increasing their rbf_s. If memory-intensive task has lowest priority, then its $MI(t)$ does not effect other tasks in the server.

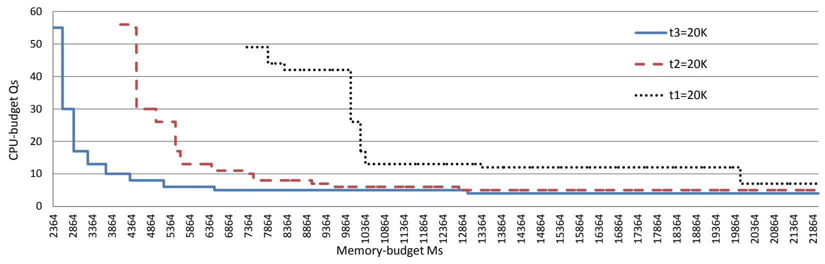


Figure 11.5: Effect of High- and low-priority memory-intensive task on M_s and Q_s using private banks only

Experiment 3: Impact of different server periods on server-budgets

This experiment is performed for different server periods (ranging from $20ms$ till $80ms$) for the task set presented in Table 11.1. The results are presented in Figure 11.6, where x-axis presents the server's memory-bandwidth utilization ($M_s/P_s \times 64 \times 1000/(1024 * 1024)$ in MB/sec), and y-axis presents CPU utilization% ($Q_s/P_s \times 100$).

In our results, sometimes the longer server period has a smaller CPU utilization when memory utilization is small as compared to the shorter server periods (i.e. in Figure 11.6, $P_s = 80ms$ has smaller CPU utilization at point $2,5MB/s$ than for $P_s = 40ms$, and for $P_s = 20ms$ the system needs more that 100% CPU utilization thus not schedulable). However, when the memory utilization is increased, the shorter server periods need a smaller CPU utilization.

In general using traditional server-based scheduling, a longer server period results in an increase in the blackout duration that requires bigger CPU budget to schedule the server [4]. However, from the memory perspective of the MRS, looking at equation 11.7, the increase of memory budget has a big impact on $A(i, t)$ function. Bigger value of M_s, due to the floor function, decreases the value of $A(i, t)$ in equation 11.7, which intern decreases the CPU budget requirement to schedule the server. We observe in Figure 11.6 that at smaller values of memory utilization (i.e. $2.5MB/s$), the longer server period ($P_s = 80ms$) has smaller CPU utilization because the impact of equation 11.7 dominates. In other cases where memory utilization has increased, the blackout duration effect is dominating. Thus we conclude that the behaviour of MRS differs from the behaviour of traditional servers.

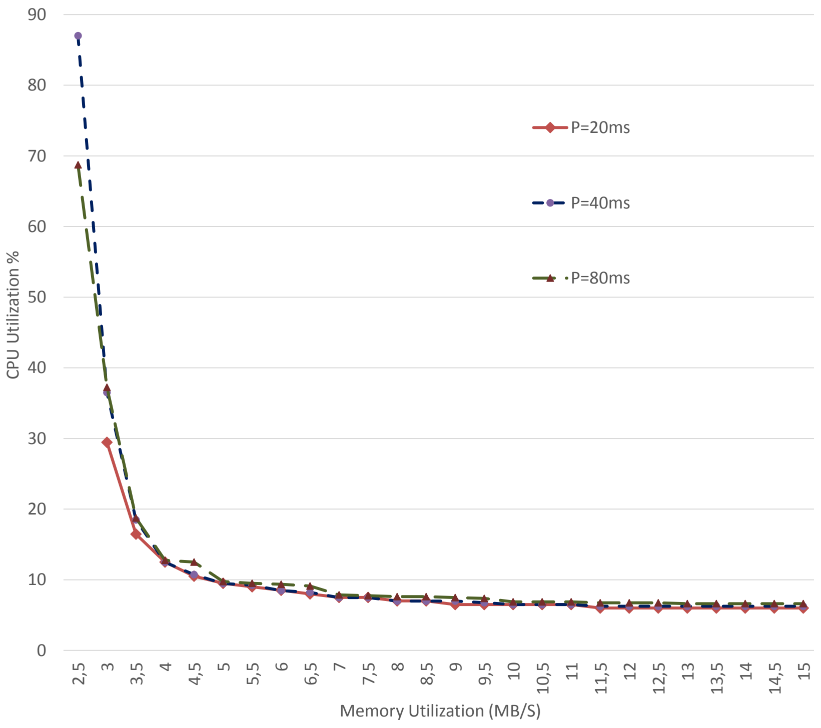


Figure 11.6: Impact of different server periods on budgets

Candidate interfaces for the MRS

We present an example consisting of four servers executing on two different cores with task sets presented in Table 11.2. We present results of using only 2 cores because of the space limitations. The server periods is assigned as half of its shortest task period [4], and multiple candidate interfaces for CPU and memory budgets are computed using local analysis as presented in Figure 11.7. It is up to the designer to select the suitable candidate interface depending on both global schedulability tests. The global schedulability tests for CPU- and memory budgets can be performed using equations of Section 11.4.2.

During the subsystem development phase, selecting the optimal interface including both budgets is not feasible without providing the details of the other subsystems' interfaces, which is not possible. To overcome this problem, we propose a similar solution as presented in [28], i.e., using a finite set of CPU and memory budgets values (called *candidate interfaces*). A candidate interface is chosen when the CPU budget changes as a function of changing the memory budget (see Figure 11.7). These candidate interfaces can be used later in the subsystems integration phase. It is not straightforward to find an optimal interface for the MRS, since a decrease in one budget value results in an increase in the second budget value. Finding optimal candidate interface selection for the MRS is left for the future.

Tasks	τ_0	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6
`Server`	`S0`	`S0`	`S1`	`S2`	`S2`	`S3`	`S3`
`Prio`	H	L	H	H	L	H	L
`T`	40	80	160	80	160	240	240
`WCET`	2	4	8	4	10	8	8
`CM`	$20K$	$40K$	$80K$	$40K$	$60K$	$60K$	$60K$

Table 11.2: Task properties.

Discussion

From experiment 1, we observe that Q_s decreases significantly at the start with the increase of memory budget M_s. And after a certain value, more increase in M_s does not effect the value of Q_s much. This helps in selecting the suitable values for Q_s and M_s. We also observe (see Exp. 3) that the behaviour of MRS is not similar to the traditional server. The change in server period has different impacts on CPU and memory budgets.

Core 0								Core 1							
S0				S1				S2				S3			
Priority	P	Q	M	Priority	P	Q	M	Priority	P	Q	M	Priority	P	Q	M
High	20	17	20001	Low	80	50	40001	High	40	35	35001	Low	120	77	60001
		16	22132			49	45988			34	35330			76	65580
		15	24400			48	52791			33	37597			75	72382
		14	26667			47	59593			32	39865			74	79185
		12	26668			46	66396			31	42132			73	85988
		11	31498			45	73199			30	44400			72	92791
										29	46667			71	99593
										24	46668			70	106396
										23	51294			69	113199

Figure 11.7: Multiple candidate interfaces for global schedulability analysis. Empty cell means not schedulable.

The presented analysis in this paper is pessimistic. We can identify some factors that contribute to the pessimism. The biggest reason of pessimism is our consideration of worst case M_s depletion where all memory requests are generated at the start of the server period. As a result M_s depletes and the remaining Q_s is discarded (see Figure 11.3). However, it could not be the case in reality. To improve the analysis, we should look at the start of task periods. We leave this improvement as a future work.

Over-provisioning the budget: typically in server based scheduling, if possible, we can over-provision the resource to get the shorter response times of the tasks. We calculated Q_s without considering memory requests (CM_i is 0 for all tasks). We get a constant value for Q_s which can be considered as a reference point. By adding memory requests CM_i, the value of Q_s will increase slightly. In order to get the shorter response times for the tasks, the value of Q_s can be increased (budgets can be over-provisioned), as long as the global schedulability of the system is satisfied.

11.6 Conclusion

The multi-resource server (MRS) approach has been proposed to address composability of independently developed real-time subsystems executed on a multicore platform. The memory-bandwidth is added as an additional server-resource to bound memory interference from other servers executing concurrently on other cores thus to provide predictable performance of multiple subsystems. Consequently, tasks within the MRS execute provided with both CPU- and memory-budgets. In this paper, we have presented a compositional analysis framework for the MRS including a complete and composable local and global analysis. For memory interference, we have safely bounded the memory contention for DDR DRAM memory controller that are commonly used in COTS multicore architectures. Further, we have performed an experimental study to investigate the correlation between the server budgets and the impact of different server periods on server-budgets. We also provide indications to find the candidate interfaces. Finding optimal interfaces for MRS is an open issue.

We have explored the source of pessimism in our analysis and in future we intend to remove some pessimism from the analysis. Another future direction is to find an algorithm to calculate the optimum budgets for both resources of the MRS and to find a smart online algorithms to assign the unused capacity of one resource to another server to improve the overall average response times.

Bibliography

[1] R. Rettberg and R. Thomas. Contention is no obstacle to shared-memory multiprocessing. *Commun. ACM*, 29(12):1202–1212, December 1986.

[2] J. P. Lehoczky, L. Sha, and J.K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proc. 8th IEEE Real-Time Systems Symposium (RTSS' 87)*, pages 261–270, December 1987.

[3] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems*, 1(1):27–60, June 1989.

[4] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proc. 24th IEEE Real-Time Systems Symposium (RTSS' 03)*, pages 2–13, December 2003.

[5] M. Behnam, R. Inam, T. Nolte, and M. Sjödin. Multi-core composability in the face of memory bus contention. *ACM SIGBED Review, Special Issue on 5th Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, 10(3), 2013.

[6] R. Inam, N. Mahmud, M. Behnam, T. Nolte, and M. Sjödin. The Multi-Resource Server for predictable execution on multi-core platforms. In *Proc. 20th IEEE Real-Time Technology and Applications Symposium (RTAS' 14)*, pages 1 – 11, April 2014.

[7] 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, September 2004.

[8] G. Lipari and E. Bini. Resource Partitioning among Real-time Applications. In *Proc. of the 15th Euromicro Conf. on Real-Time Systems (ECRTS' 03)*, pages 151–158, July 2003.

[9] I. Shin, A. Easwaran, and I. Lee. Hierarchical Scheduling Framework for Virtual Clustering of Multiprocessors. In *Proc. of the 20th Euromicro Conf. on Real-Time Systems (ECRTS' 08)*, pages 181–190, July 2008.

[10] J. Rosen, A. Andrei, P. Eles, and Z. Peng. Bus Access Optimization for Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. In *Proc. 28th IEEE Real-Time Systems Symposium (RTSS' 07)*, pages 49–60, December 2007.

[11] A. Schranzhofer, R. Pellizzoni, J.-J. Chen, L. Thiele, and M. Caccamo. Worst-case Response Time Analysis of Resource Access Models in Multi-core Systems. In *Proc. of the 47th Design Automation Conference (DAC '10)*, pages 332–337. ACM, 2010.

[12] B. Akesson, K. Goossens, and M. Ringhofer. Predator: A Predictable SDRAM Memory Controller. In *Int'l Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 251–256, September 2007.

[13] S. Schliecker and R. Ernst. Real-time Performance Analysis of Multiprocessor Systems with Shared Memory. *ACM Transactions in Embedded Computing Systems*, 10(2):22:1–22:27, January 2011.

[14] D. Dasari and B. Anderssom and V. Nelis and S.M. Petters and A. Easwaran and L. Jinkyu . Response Time Analysis of COTS-Based Multicores Considering the Contention on the Shared Memory Bus. In *Proc. of the IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom '11)*, pages 1068 – 1075, November 2011.

[15] R. Pellizzoni and A. Schranzhofer and J.-J.Chen and M. Caccamo and L. Thiele. Worst Case Delay Analysis for Memory Interference in Multicore Systems. In *Proc. of the Conference on Design, Automation and Test in Europe (DATE' 10)*, pages 759–764, 2010.

[16] Z-P. Wu, Y. Krish, and R. Pellizzoni. Worst case analysis of DRAM latency in multi-requestor systems. In *Proc. 34th IEEE Real-Time Systems Symposium (RTSS' 13)*, pages 372–383, December 2013.

[17] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. R. Rajkumar. Bounding memory interference delay in COTS-based multicore systems. In *Proc. 20th IEEE Real-Time Technology and Applications Symposium (RTAS' 14)*, pages 145–154, April 2014.

[18] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory- access Control in Multiprocessor for Real-Time Systems with Mixed Criticality. In *Proc. of the 24th Euromicro Conf. on Real-Time Systems (ECRTS' 12)*, July 2012.

[19] R. Inam, M. Behnam, and M. Sjödin. Worst case delay analysis of a DRAM memory request for COTS multicore architectures. In *Seventh Swedish Workshop on Multicore Computing (MCC' 14)*, November 2014.

[20] Micron. 2Gb DDR3 SDRAM.

[21] K.J. Nesbit, N. Aggarwal, J. Laudon, and J.E. Smith. Fair queuing memory systems. In *In International Symposium on Microarchitecture (MICRO)*, 2006.

[22] H. Yun, R. Mancuso, Z-P. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *Proc. 20th IEEE Real-Time Technology and Applications Symposium (RTAS' 14)*, April 2014.

[23] L. Sha, J.P. Lehoczky, and R. Rajkumar. Solutions for some Practical problems in Prioritised Preemptive Scheduling. In *Proc. 7th IEEE Real-Time Systems Symposium (RTSS' 86)*, pages 181–191, December 1986.

[24] J. Liedtke, H. Härtig, and M. Hohmuth. OS-controlled cache predictability for real-time systems. In *Proc. 3rd IEEE Real-Time Technology and Applications Symposium (RTAS' 97)*, pages 213–224, June 1997.

[25] D. Hardy and I. Puaut. Estimation of cache related migration delays for multi-core processors with shared instruction caches. In *17th International Conference on Real-Time Networks and Systems (RTNS' 09)*, October 2009.

[26] A. Bastoni, B. B. Brandenburg, and J. H. Anderson. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. In *6th Annual workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT' 10)*, July 2010.

[27] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Proc. 19th IEEE Real-Time Technology and Applications Symposium (RTAS' 13)*, pages 55–64, 2013.