Mälardalen University Licentiate Thesis
No.67

Introducing a Memory Efficient Execution Model in a Tool-Suite for Real-Time Systems

Kaj Hänninen

2006

Department of Computer Science and Electronics
Mälardalen University
Västerås, Sweden

Copyright © Kaj Hänninen, 2006
ISSN 1651-9256
ISBN 91-85485-22-5
Printed by Arkitektkopia, Västerås, Sweden
Distribution: Mälardalen University Press

Abstract

This thesis shows how development of embedded real-time systems can be made more efficient by introduction of a memory efficient execution model in a commercial development suite. To identify the need of additional support for execution models in development tools, the thesis investigate by a series of interviews, the common requirements in development of industrial embedded real-time systems. The results indicate that there exists functionality in industrial systems that could be more efficiently implemented in other execution models than the currently supported ones. The thesis then presents how use of multiple execution models (hybrid scheduling) can reduce processor utilization in real-world applications. Furthermore, the thesis presents an integration of a memory efficient execution model in an industrially used real-time operating system. In addition, the thesis describes an efficient technique to analyze memory consumptions of functionality executing under the introduced execution model.

Embedded computers play an important role in peoples everyday life. Nowadays, we can find computers in product such as microwave ovens, washing machines, DVD players, cellular phones and cars, to mention a few examples. For example, a modern car may have more than 70 embedded control units handling functionality such as airbags, anti-lock braking, traction control etc. In addition, there is a clear trend indicating that the amount of computer controlled functionality in products will continue to increase.

Many of today's embedded systems are resource constrained and the software for them is developed for a few execution models. Even though researchers have proposed a large number of different execution models for embedded real-time systems, in practice however, only a few of the proposed execution models are supported in industrial development tools. This implies that developers often force fit functionality to be executed under these models, resulting in poor resource utilization and increasing complexity in software.

To the memory of Stissen

Preface

The work presented in this thesis has been sponsored by KK-Stiftelsen (the Swedish Knowledge Foundation) within the graduate school SAVE-IT .

I wish to thank the following people for their excellent supervision, guidance, encouragement etc.: Mikael Nolin, Jukka Mäki-Turja, Christer Norström, Kurt-Lennart Lundbäck, John Lundbäck, Hjördis Lundbäck, Mats Lindberg, Mikael Arvids, Harriet Ekwall and all research colleagues at Mälardalen Real-Time Research Centre (MRTC).

I wish to express my deepest gratitude to: CG, Pastisch, M'Morris, Azze, Krutte, Chibi, Esa, Peter and the PSBs.

Kaj Hänninen
Västerås, September, 2006

List of Publications

Publications included in this thesis

Paper A: Kaj Hänninen, Jukka Mäki-Turja, Mikael Nolin, *Present and Future Requirements in Developing Industrial Embedded Real-Time Systems - Interviews with Designers in the Vehicle Domain*, In Proceedings of the 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, Potsdam, Germany, March, 2006.

Paper B: Jukka Mäki-Turja, Kaj Hänninen, Mikael Nolin, *Efficient Development of Real-Time Systems Using Hybrid Scheduling*, In Proceedings of the International Conference on Embedded Systems and Applications, Las Vegas, USA, June, 2005.

Paper C: Kaj Hänninen, John Lundbäck, Kurt-Lennart Lundbäck, Jukka Mäki-Turja, Mikael Nolin, *Efficient Event-Triggered Tasks in an RTOS*, In Proceedings of the International Conference on Embedded Systems and Applications, Las Vegas, USA, June, 2005.

Paper D: Kaj Hänninen, Jukka Mäki-Turja, Markus Bohlin, Jan Carlson, Mikael Nolin, *Analysing Stack Usage in Preemptive Shared Stack Systems*, MRTC report ISSN 1404-3041 ISRN MDH-MRTC-202/2006-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, July, 2006. A version of this paper has been accepted for publication at RTSS, Rio de Janeiro, Brazil, December, 2006.

Other publications by the author

- Jukka Mäki-Turja, Mikael Nolin, Kaj Hänninen, *Towards Efficient Development of Embedded Real-Time Systems, the Component Based Approach*, The 2006 International Conference on Embedded Systems and Applications, Las Vegas, USA, June, 2006
- Kaj Hänninen, Jukka Mäki-Turja, Mikael Nolin, *Investigation of Industrial Requirements in Development of Embedded Real-Time Systems*, MRTC report ISSN 1404-3041 ISRN MDH-MRTC-185/2005-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, August, 2005
- Kaj Hänninen, Jukka Mäki-Turja, Mikael Nolin, *Industrial Requirements in Development of Embedded Real-Time Systems -Interviews with Senior Designers*, Work-in-Progress Session of the 17th Euromicro Conference on Real-Time Systems, Palma de Mallorca, Spain, July, 2005
- Kaj Hänninen, John Lundbäck, Kurt-Lennart Lundbäck, Jukka Mäki-Turja, Mikael Nolin, *Introducing Resource Efficient Event-Triggered Tasks in an RTOS*, MRTC report ISSN 1404-3041 ISRN MDH-MRTC-170/2005-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, February, 2005
- Jukka Mäki-Turja, Kaj Hänninen, Mikael Nolin, *Response Times in Hybrid Scheduled Systems*, MRTC report ISSN 1404-3041 ISRN MDH-MRTC-169/2005-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, February, 2005
- Kaj Hänninen, Jukka Mäki-Turja, *Component technology in Resource Constrained Embedded Real-Time Systems*, Technical Report, MRTC, March, 2004
- Toni Riutta, Kaj Hänninen, *Optimal Design*, Master's thesis, Mälardalen University, Department of Computer Science and Engineering, February, 2003

Contents

I

Thesis

Chapter 1

Introduction

Throughout the years, software development for embedded computer systems has undergone changes. Development processes have been modernized and refined to fulfil the evolving requirements on applications. High level programming languages (e.g. C, C++) have been adopted as alternatives to the traditional low level programming languages such as assembler. A more recent trend is the introduction of component- and model-based engineering in the embedded community.

However, very little effort has been done to encapsulate novel execution models in commercial operating systems and development tools. Even though researchers within the real-time community have provided a large number of different execution models, only a few of them are actually used in industrial systems. Many of the research project addressing co-existence of several different execution model within a single computer system, has ended up as pure academic work.

Taking advantage of the novel execution models provided by the research community, the development of embedded real-time systems can be made more efficient, with respect to memory and processor utilization, while fulfilling requirements on predictable use of resources. This thesis shows how an efficient execution model can be integrated as support in an industrial tool-suite for development of dependable real-time systems. The integrated execution model, denoted SSX, enables stack sharing among tasks for efficient use of memory. Furthermore the thesis presents methods that allow the execution model to be analyzed with respect to timing and memory consumption, making the execution model usable in an industrial setting.

1.1 Embedded real-time systems

In conventional computer systems, functional correctness is of primary concern. Real-time systems however, emphasize both functional and timing requirements in computing. One important and commonly used timing requirement in real-time systems is called: the deadline. The deadline is a representation of the latest point in time that a functionality must be finished. In fact, the deadline attribute is of such importance that real-time systems are categorized into either hard of soft systems, based on whether deadlines are critical or not. In hard real-time systems it is required that all deadlines are met, whereas soft systems allow certain missed deadlines. In addition to the deadline attribute, other types of timing requirements are applicable for real-time systems, see e.g., [2, 4, 7] for more examples.

To investigate whether timing requirements can be fulfilled or not in a real-time system, the system must lend itself to analysis. In essence, this implies that both the hardware and software should be analyzable with respect to timing attributes. In addition, real-time systems often interact with the environment by sensors and actuators. In many of the environments, safety critical situations can occur, implying that dependability issues have to be considered in development of such systems.

1.2 Execution models

Computer systems, desktop types or embedded ones, execute processes, threads or tasks, according to one or more execution models. In general, an execution model can be seen as methods that provide ways to execute tasks or carry out functionality in computer systems. The execution model in a real-time system can be composed of, for example, a scheduling technique, a task- and memory-model. Research in scheduling techniques for real-time systems has resulted in a large number of different scheduling algorithms, see e.g., [3, 1, 7, 2, 4, 6, 5, 9], many of them are developed for a conventional memory model and a specific task model. In addition, many real-time systems involve concurrent activities, implying that the execution model is one of the fundamental parts in realizing a system of parallel (seemingly parallel in uni-processor systems) executing tasks. In distributed computer system with functionality executing over several nodes, many execution models might be involved in realizing a functionality, e.g., execution models for the computing nodes and execution models for the communications network.

This thesis addresses the integration of a predictable and memory efficient execution model, denoted SSX, in a commercially available tool-suite. The SSX model is only one of several execution models proposed by the research community. The model is attractive since it permits run-time stack sharing among tasks, hence reducing the amount of memory needed in many type of applications. Furthermore, the SSX model can be implemented with very little run-time overhead in an operating system.

1.3 Execution models in development

In development of embedded real-time systems, a developer must choose an execution model to realize a functionality. For instance, using the Rubus tool-suite [8], a developer can choose from three supported execution models (i) a cyclic time-triggered static execution model, (ii) a dynamically scheduled background executed model or, (iii) dynamically scheduled event-triggered high priority model for interrupts. Every execution model has its benefits and drawbacks. For example, a cyclic time-triggered static execution model is predictable and reproducible, hence often used for safety critical functionality. However, since real-time systems often require reaction to external events, the time-triggered model requires polling of the events. Polling events often waste processor time, since polling is performed regardless of whether an event has occurred or not. On the other hand, an execution model supporting dynamic scheduled tasks might be less resource demanding than the time-triggered model, but it is more difficult to reproduce an execution (for testability) of the system, under a dynamic execution model.

Undoubtedly, choosing suitable execution models in development of a system is associated with a lot of issues, some of them arising from application requirements e.g. the type of functionality and their temporal requirements, others from the activities in a development process, e.g., availability of analysis support. It is likely that the requirements, the available tool support, implementation, testing and maintainability aspects as well as target platform etc, all guides the choice of execution models. In addition, many operating systems only provide a few (commonly one or two) execution models to be used in development.

It's obvious that the choice of execution model greatly affects development, in that all design decisions ultimately must adhere to the support provided by the execution model. With the increasing amount of diverging type of functionality, ranging from safety critical functionality to non critical functionality such as information and entertainment, new and more suitable execution models must be supported by commercial operating systems and development tools. Restricting the developers to use one specific execution model (e.g. a static cyclic model) for all functionality, increases the complexity of systems since developers are forced to come up with solutions adhering to the specific execution model. In addition, force fitting all functionality to be executed under one execution model often results in systems where the available resources, such as processing time and memory, are unnecessary over-allocated. Hence, as the need for new features in products increases, so do the need for better usage of the limited computational resources, by resource efficient and predictable execution models.

1.4 Outline of thesis

The reminder of this thesis is outlined as follows.

Part I
Chapter 2 outlines the methodology. It describes the background and the addressed problems. Furthermore, the chapter outlines the research questions and presents the contribution of the thesis.
Chapter 3 concludes the thesis.
Part II
The second part of this thesis contains publications with detailed descriptions of the research carried out within this thesis.

Chapter 2

Research framework

This chapter presents the research framework used in this thesis. It gives information of the research methodology, background and problem formulation. The research question addressed in this thesis is presented and motivated. The chapter ends with a presentation of the contributions of the thesis.

2.1 Methodology

The research forming this thesis has been performed in close cooperation with industry. Arcticus Systems AB, an OS and tool developer with recognized competence in design of real-time applications, has been the main industrial partner.

The research problems addressed in this thesis are mainly based on a series of interviews with senior designers in the heavy vehicle domain. Answering the research questions has mainly been done by prototyping. Prototypes of Rubus OS supporting multiple execution models, and an efficient application to compute memory requirements for an execution model (the SSX model), has been developed. Validation of the research requires the prototypes to be integrated as support in sharp releases of the Rubus tool-suite. The integration has been initiated and is currently an ongoing activity. The next generation of the Rubus tool-suite will support a memory efficient stack sharing execution model (SSX model). Furthermore, the tool-suite will have a timing and memory analysis application for the SSX model integrated.

2.2 Background and motivation

Computer scientists are constantly proposing new theories to improve development of real-time systems. For example, the research community has provided a large numbers of different execution-models, however, only a few of these models has gained real interest in the industrial domain. In general, very little of the theories that allows predictable and resource efficient integration of multiple execution-models has been adopted by industry. The reason for this, we believe is, that there has been little effort done to encapsulate these theories by supporting development tools and techniques.

The aim of this thesis is to study the possibility to make development of embedded real-time systems more efficient by introducing support for a predictable and memory efficient execution model in a tool-suite for development of embedded real-time system.

2.3 Problem description

With the large number of different execution models, provided by the research community, an investigation of common requirements within the intended domain is required. It is required since many of the proposed execution models are developed in academia and often without considering actual real-world requirements. There are several real-world requirements that put restrictions on execution models and hence restrict the number of suitable models for an intended domain. For example, development processes and safety issues as well as target platforms might put specific requirements on execution models.

Moreover, a predictable integration of a novel execution model for development of dependable systems, require careful design and consideration of safety issues when implemented in an operating systems. This is especially important when several execution models must co-exist in the same operating system. In addition, many of the academic execution models only provide analyzability of timing properties. However, in real-world applications it is likely that both timing properties and memory consumptions should be analyzable. Hence, an efficient execution model for industrial development might need to be analyzable with respect to timing and memory consumption, while co-existing with other execution models. This might require development of novel analysis methods.

2.4 Research questions

To address the research problem defined in the previous section, the following research questions was formulated.

What are common requirements in development of embedded real-time systems?

(Q-1)

In order to integrate support for an efficient and suitable execution model in development tools, the common requirements within the intended domain should be investigated. This question will give an indication of the main properties that an execution model should support. The question is motivated by the fact that many systems are resource constrained and that the architectures can differ between different systems. For example, within the heavy vehicle domain, the systems often consist of a few electronic control units (ECU) with a lot of functionality (hundreds of tasks) within each ECU. On the other hand, within the automotive domain, the systems are developed with a large number of ECUs, where each ECU typically has a dedicated functionality. This might affect requirements on the execution models, in the sense that an execution model must be efficient due to resource constraints and predictable for different type of functionality to execute on the same target.

How can development using multiple execution models (hybrid static, dynamic) be efficient with respect to processor utilization?

(Q-2)

This question will answer how development of embedded systems, using static and dynamic execution models, can be made more efficient with respect to processor utilization. The question is motivated because many embedded systems have very limited amount of processing time, and by the fact that the amount of functionality in embedded systems are increasing. The question will answer how the available processing time can be efficiently used in a system with static and dynamic execution models.

How can the SSX model be integrated to co-exist with other execution models in a real-time operating system?

(Q-3)

This question will answer how the SSX model can be integrated to co-exist with other execution models in an operating system used in development of industrial real-time systems. The question is motivated by the fact that many embedded systems contain safety critical functionality. Hence, when introducing a new execution model, it must be guaranteed that the new model do not interfere with the existing ones in the sense that it affect the safety of the systems.

How can the run-time memory consumption within the SSX model be analyzed?

(Q-4)

This question addresses analyzability of memory requirements, specifically the stack memory requirements, for functionality executing under the SSX model. The question will answer how to efficiently predict memory requirements for functionality executing under the execution model. It is motivated by the fact that most execution models only provide timing analysis of functionality. Very little effort has been done to analyze memory consumption of execution models.

2.5 Contribution

This section addresses the research questions and describes the contributions of the author.

What are common requirements in development of embedded real-time systems?

(Q-1)

Paper A addresses Q-1. The paper presents a series of interviews, aiming to investigate requirements in development of heavy vehicles. The paper describes both current and future requirements.

How can development using multiple execution models (hybrid static, dynamic) be efficient with respect to processor utilization?

(Q-2)

Paper B addresses Q-2. The paper shows how a hybrid, static and dynamic, scheduled system can be made more efficient by moving functionality from the static scheduled part to the dynamically scheduled part while guaranteeing analyzability of timing properties.

How can the SSX model be integrated to co-exist with other execution models in a real-time operating system?

(Q-3)

Paper C addresses Q-3. The paper describes an integration of the SSX model in a commercial operating system called Rubus. The SSX model is motivated by requirements described in paper A and the fact that the model lend itself to timing analysis, as described by paper B.

How can the run-time memory consumption within the SSX model be analyzed?

(Q-4)

Paper D addresses Q-4. The paper describes an exact and an approximate method to establish an upper bound on maximum stack usage in preemptive shared stack systems. The described methods can be applied to, for example, tasks executing under the SSX model.

The following describes the authors contributions in detail.

Paper A: Kaj Hänninen, Jukka Mäki-Turja, Mikael Nolin, *Present and Future Requirements in Developing Industrial Embedded Real-Time Systems - Interviews with Designers in the Vehicle Domain*, In Proceedings of the 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, Potsdam, Germany, March 2006.

Kaj was the main author of the paper and has been the involved in all parts of the work. He was responsible of preparing the study and establishing the research questions. He coordinated the interviews and analyzed the results. He was also responsible of reporting the results.

Paper B: Jukka Mäki-Turja, Kaj Hänninen, Mikael Nolin, *Efficient Development of Real-Time Systems Using Hybrid Scheduling*, In Proceedings of the International Conference on Embedded Systems and Applications, Las Vegas, USA, June 2005.

Kaj has been involved and provided the real-world information for the case study part of the paper.

Paper C: Kaj Hänninen, John Lundbäck, Kurt-Lennart Lundbäck, Jukka Mäki-Turja, Mikael Nolin, *Efficient Event-Triggered Tasks in an RTOS*, In Proceedings of the International Conference on Embedded Systems and Applications, Las Vegas, USA, June 2005.

Kaj was the main author of the paper and has been involved in all parts of the work. He was responsible of implementing the execution model in Rubus and evaluating the implementation.

Paper D: Kaj Hänninen, Jukka Mäki-Turja , Markus Bohlin, Jan Carlson, Mikael Nolin, *Analysing Stack Usage in Preemptive Shared Stack Systems*, MRTC report ISSN 1404-3041 ISRN MDH-MRTC-202/2006-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, July, 2006.

Kaj was the main author of the paper. He initiated and coordinated the work and did the background research on related work. He was also responsible for parts of the implementation and did the evaluation of the approximate method.

Chapter 3

Conclusion

This thesis addressed the introduction of an efficient execution model in a tool-suite for development of industrial embedded real-time systems. The thesis showed, by an investigation of industrial requirements, that many systems are developed using only a few execution models and that introduction of novel execution models could make development of industrial systems more efficient.

An integration of an efficient stack sharing execution model (SSX) in a real-time operating system was presented. The integration was presented in detail to highlight common issues in implementing support for multiple execution models in a real-time operating system. Furthermore, it was shown that the integrated execution model could be analyzed for both timeliness and memory consumption. A case study of an industrial system showed that reductions in processor utilization could be achieved, by using the integrated execution model.

In essence, this thesis showed that introduction of additional execution models in industrial development tools, can make development of real-time systems more efficient.

Bibliography

[1] N. Audsley, A. Burns, M. Richardson, and A. Wellings. Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *IEEE Workshop on Real-Time Operating Systems*, 1992.

[2] G.C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic Publishers, 1997. ISBN 0-7923-9994-3.

[3] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

[4] J. Liu. *Real-Time Systems*. Prentice Hall, 2000. ISBN 0-13-099651-3.

[5] K. Sandström, C. Eriksson, and G. Fohler. Handling interrupts with static scheduling in an automotive vehicle control system. In *Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications*, 1998.

[6] B. Sprunt, L. Sha, and J.P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems Journal*, 1(1), 1989.

[7] J.A. Stankovic, M. Spuri, K. Ramamritham, and G.C Buttazzo. *Deadline Scheduling for Real-Time Systems, EDF and Related Algorithms*. Kluwer Academic Publishers, 1998. ISBN 0-7923-8269-2.

[8] Arcticus systems. Web page, http://www.arcticus-systems.se.

[9] J. Xu and D.L Parnas. Scheduling processes with release times, deadlines, precedence and exclusion relations. *IEEE Transaction on Software Engineering*, 16(3), 1990.

II

Included Papers

Chapter 4

Paper A: Present and Future Requirements in Developing Industrial Embedded Real-Time Systems - Interviews with Designers in the Vehicle Domain

Kaj Hänninen, Jukka Mäki-Turja, Mikael Nolin
In Proceedings of the 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, Potsdam, Germany, March, 2006.

Abstract

In this paper, we aim at capturing the industrial viewpoint of todays and future requirements in development of embedded real-time systems. We do this by interviewing ten senior designers at four Swedish companies, developing embedded applications in the vehicle domain. This study shows that reliability and safety are the main properties in focus during development. It also shows that the amount of functionality has been increasing in the examined systems. Still the present requirements are fulfilled using considerably homogenous development methods. The study also shows that, in the future, there will be even stronger requirements on dependability and control performance at the same time as requirements on more softer and resource demanding functionality will continue to increase. Consequently, the complexity will increase, and with diverging requirements, more heterogeneous development methods are called for to fulfil all application specific requirements.

4.1 Introduction

There is an increasing trend towards software solutions in embedded systems. Replacing mechanical functionality with computer-controlled solutions gives opportunities for more advanced and more flexible functionality, e.g., anti-lock braking, traction control etc. Over the years, a large number of publications, e.g., [1][2][3][4][5][6][7][8] has addressed design issues, embedded application trends or requirements in development of industrial embedded systems. Möller et al [4] present the industrial requirements, both technical as well as process related requirements, on component technologies in the heavy vehicle domain. Åkerholm et al [8] presents an investigation concerning classification of quality attributes for component technologies in the vehicle industry. The investigation show that dependability characteristics (safety, reliability and predictability) are considered as the most important ones. Koopman [3] presents attributes of four different types of embedded systems (signal processing systems, mission critical and distributed control systems and consumer electronic systems). Koopman addresses requirements, life-cycle support and business models in development of embedded systems. Graaf et al [1] presents an industrial inventory of seven companies developing embedded software products. Their inventory of state of practice addresses requirements engineering and architectural issues such as design and analysis. The inventory covers companies from many different domains, e.g., developers of mobile phones and consumer electronics, distributed data management solutions etc. In this paper, we investigate the industrial requirements in the vehicle domain, especially requirements related to real-time issues on a high overall level, such as safety and reliability requirements of embedded application/products, as well as on a lower technical level, such as choice of operating system (OS) and execution models. The study was performed as a series of interviews with ten senior designers at four Swedish companies. Specifically, we address the following questions: Q1. What characterise the embedded applications? Q2. What are the designers concern on application properties such as safety, maintainability, testability, reliability, portability and reusability? Q3. How are the applications verified/analysed? Q4. What are the considerations in choosing an OS, and execution model? Q5. What resources are considered as constrained in the systems, and to what degree? Q6. What kind of tool support is needed in the development of future systems? Q7. What are the designers experiences of software components, i.e., component based development?

The aim of this work is foremost to explore and describe the current and future industrial requirements as perceived by the senior designers. The paper is organised as follows. In Section 4.2, we describe the framework used in the study of the requirements. In Section 4.3, we describe the results of the conducted interviews. In Section 4.4, we address the main questions of the study and discuss our observations of the interviews. In Section 4.5, we conclude our investigation. The paper ends, in Section 4.6, with a discussion concerning verification of the presented results.

4.2 Investigation setup

For this study, we adopted the investigation framework described by Robson [9]. According to the framework, both the purpose of a study and the theory guiding the study should form as guidance when developing the actual research questions. The substance and the form of the research questions form the basis when deciding on suitable investigation method and sampling strategy.
Purpose: The main objective of this study was to investigate the typical set of industrial requirements in development within the embedded control community in the vehicle domain. The results are expected to form a foundation for further research on tool support, design, analysis and synthesis of embedded real-time systems using multiple execution models.
Theories: Our experience is that there has been little effort done to encapsulate novel theories by supporting development tools and techniques in the industrial domain. Traditionally, the development of these systems tends to focus on the safety critical parts, which constitutes a small fraction of the total system functionality. Homogenous development methods are often used for both the safety-critical and the non-critical functionality in the systems. This results in unnecessary complex designs and over utilised systems, where valuable resources such as processing time and memory resources are wasted. We believe that there is a need for more sophisticated development support compromising of additional tool support and more domain and application specific development platforms, resulting in a more heterogeneous development environment and resource efficient run-time structure. The development platform should aim at handling complexity by relieving the developer of too low details while preserving predictability for core functionality as well as flexibility for less critical functionality in the run-time structure.
Questions: We compromised upon a set of quantitative and qualitative, closed and open-ended questions. The main purpose of the quantitative questions was to facilitate analysis of importance among application properties.
Data collection: Due to the substance and the form of the research questions,

the study was conducted as face-to-face interviews using a questionnaire. A pilot study was performed at an OS and development tool vendor. The purpose of the pilot study was to refine the data collection plans and to evaluate the feasibility of the chosen data collection method. The structure of the questionnaire was refined and additional questions were added, as a result of the pilot study.
Sampling: In this study, we use purposive non-probability samples, i.e., the samples are selected as to interest (we do not make generalisation to any population beyond the samples). Four successful and renowned companies in the Swedish vehicle domain were participating in the study. The samples represent both subcontractors as well as own equipment manufacturers. Moreover, the selection is a representative subset of both off-road and road vehicles. The companies range from small and medium-sized enterprises to large corporate groups. The thorough examination of the applications and development processes require us for secrecy reasons to refer the companies as A, B, C and D. Ten software designers with several years of experience from development of control systems for embedded real-time systems participated in the study. For preparation reasons, the questionnaire was mailed in advance to each interviewee.
Analysis: Upon agreement with the interviewees, each interview was tape-recorded. The recordings and notes taken during the interviews were interpreted and analysed both individually and at group basis. We did however not use any specific software package to interpret or analyse the collected data.
Biases: Several factors may introduce unwanted biases in a real-world study. For example, recording interviews may affect the respondent, welcoming or sharing the respondents' views may affect the interview, and so on. To avoid or at least minimise possible biases, we followed recommendations given in [10][9][11] about how to construct questionnaires and conduct face-to-face interviews in real-world situations. It is our experience that the research questions were easily understood and similarly interpreted by the interviewees, and that the recording had no or very little effect on the respondents and the outcome of the interviews.

4.3 Investigation results

In the following section, we describe: (i) Real-time and functional characteristics of the examined applications. (ii) The interviewees concern on selected application properties. (iii) The currently used resource management policies and available execution models of the examined applications. (iv) The actual resource situation in the examined systems i.e., availability of computing resources such as CPU time and memory. (v) Some desired support in development tools, as expressed by the interviewees.

4.3.1 Application characteristics

The product volumes of the investigated applications are typically less than 1000 products per year. The applications are mainly used as control applications for various types of vehicles. In addition to control functionality, the applications typically contain functionality for information handling such as logging for diagnostic purposes and presentation of data i.e., visual interaction with the system operators. The architectures of the examined systems are of distributed character where several nodes, Electronic Control Units (ECUs), perform computations and communicate with each other mainly via CAN buses. Each ECU is usually dedicated to handle specific type of functionality e.g., an engine controller is mainly responsible for controlling engine specific functionality such as fuel injection, ignition etc. The number of ECUs in the systems has typically been increasing over the years. For example, table 4.1 shows the amount of software and the number of control units in evolution of a single product at one of the investigated companies.

Table 4.1: An example of the amount of software and the number of ECUs in a single vehicle, at company A

Year	1991	1997	2002
Lines of code	20000	55000	140000
Files (.c, .h)	50	400	700
ECUs	1	2	3

Current characteristics: The examined applications are realized by hard and soft real-time tasks. In several systems, hard real-time tasks are used to model the majority of all functionality. In extreme cases, as much as 95% of the functionality is modelled by hard real-time tasks. In addition, functionality with requirements that are neither hard nor soft, but somewhere in-between, is often modelled as hard. In context to this, the designers stress that development of hard application tasks is considered as more controllable and simpler than development of soft application tasks. In addition, several interviewees consider time-triggered systems to be the most convenient way to model hard

real-time functionality. Typical technical requirements in the examined applications include; jitter requirements and precedence relations among tasks. The timing constraints, e.g., deadlines on different functionality, can vary as much as three orders of magnitude in a single application, typically from milliseconds to several seconds. The amount of safety critical functionality varies in the investigated applications. In all of the examined applications, the control functionality is considered as being most safety critical and developed mainly using the time-triggered paradigm. Several interviewees consider their systems being I/O intensive. In some systems, as much as 30% of the available processing time and hundreds of I/O pins is used to handle I/O functionality. The I/O functionality is realised by both time and event-triggered execution models. However, it is most commonly realised using the time-triggered model, i.e., through polling. The information intensity in the investigated applications varies. In some applications, the information originates from logging and diagnostics of the systems operational conditions, whereas other applications receive and process external information that is presented to the users during operation.
Future characteristics: The interviewees believe that the information intensity and number of control functionality will increase in the future. They state that in the future both legislation and insurance reasons will force development of more sophisticated control algorithms and require an increasing amount of information to be saved for diagnostic reasons. In addition, designers from one company predicts that legislations, especially non-pollution laws, and future trends in development of vehicle engines will require better control precision. This will result in an increased transformation from open to closed loop controlling. Furthermore, some interviewees predict that functionality interacting with the environment will be developed using fewer sensors in the future and that certain conditions/states of the environment will be derived using the remaining set of sensors. Classification of functionality in Safety Integrity Levels (SIL) [12] is also believed to be an important activity in the future.

4.3.2 Functional application properties

In this section, we present the interviewees concern on the following application properties: safety, maintainability, testability, reliability, portability and reusability.
Safety: Safety is considered as a derived property originating foremost from analysis and testing. In some of the examined systems, redundancy and certain safety properties are solved outside the actual software implementation, by

physical cabling etc. The software in these systems can be overridden by mechanics in case the software malfunctions and a safety critical situation occurs.
Maintainability: Some interviewees state that the developers consider and try to facilitate future maintainability of applications. Some interviewees also state that they have very strong requirements (economical and quality) on applications being error free since withdrawing an erroneous application would be very costly due to the product volumes. There seems to be an agreement on that maintainability will have to be considered as a more important property in the future, specifically in the context of upgradeability. The lifespan of the examined systems can be several decades and customers put demand on new features and require hardware replacement parts to be available during the entire lifespan of a system. This requires applications to be well structured and easy to understand for future developers (maintainers).
Testability: Testability is stated as an important and necessary property to achieve reliability and safety. Today testing is the main technique to verify functional requirements.
Reliability: Several interviewees state that a company's reputation is very much dependent on the reliability of the delivered systems; i.e., it is considered as being of utmost importance to develop systems that actually are, and perceived by customers as, reliable. Failure in producing reliable systems is often stated to origin from erroneous requirement specifications, i.e., not from the implementation itself.
Portability: Some interviewees do not consider portability during development, simply because they seldom change hardware or OSs. Other respondents claim that portability is an increasing concern and that it is mainly facilitated by separation of hardware and software dependent functionality.
Reusability: Reusability of both soft- and hardware is an ongoing activity in all of the examined systems. However, the amount of reusable software varies in the examined systems. Some interviewees' state that reusability of architectures is not achieved until they have undergone several modifications, hence it may takes years before certain parts of architectures are actually reusable. To facilitate reusability among different systems, some of the companies have developed common software platforms. The platforms contain all common functionality and have standardised interfaces. General software components are also mentioned as reusable entities. The components are general in the sense that they are, to a large degree, application independent.
Additional properties: When asked for additional properties that are considered as important for their applications, the interviewees mentioned robustness, scalability and usability. Robustness is defined by the respondents as

'the absence of unexpected behaviour' or as 'an additional degree of reliability'. Scalability is considered in the context of development as the ability to scale systems using the available development tools. Usability of architectures is mentioned as a process related issue. In that context, the usability of architectures is said to be dependent on whether it facilitates understanding and communication between developers. All of the respondents stress the importance of architectural descriptions as means of communication between people i.e., not only as logical or structural system description.

4.3.3 Temporal application properties

This section describes the interviewees view on the temporal analysability of the applications and verification of functional/temporal behaviour. It also addresses the verification of resource utilisation in the examined applications.
Analysability and verification: Analysis of real-time properties such as response-times, jitter, and precedence relations, are commonly performed in development of the examined applications. In this context, some interviewees stress the desire of better analysis support in development tools and state that analysing a whole system with respect to temporal and spatial attributes is very difficult, sometimes even intractable. Due to the difficulties in analysing a complete system, and for upgradeability reasons, some of the examined systems are intentionally over-dimensioned with respect to processing power and memory resources. The emphasis on verification is foremost on the functional behaviour. Our experience is that the temporal attributes are not serving as direct guiding factors (albeit they are more or less considered) during development.
Functional behaviour: All of the respondents had a unanimous opinion that analysis and verification of the functional behaviour was the most important activity in the verification and analysis processes (more important than analysis and verification of temporal behaviour). The functional behaviour is mainly verified by manual and automatic module and systems tests. Failure mode and effect analysis (FMEA) are commonly performed both during development and on complete systems. Several interviewees state that source code inspection is performed among the developers and that it serves as analysis/verification of functional behaviour.
Temporal behaviour: The verification of temporal behaviour was said to have lower importance than of functional behaviour. The temporal verification of the examined systems commonly involves verification of precedence relations among functions and verifying that deadlines are met i.e., that estimated worst-case execution times holds and that calculated worst case response-times are

met.
Verification of resource utilisation: Many of the examined systems have been evolving for several years. The amount of resources, e.g., the number of control units, has been increasing over the years. Currently, all of the examined systems have more than enough processing time and available memory to perform the intended computations. Hence, verification of resource utilisation, such as memory consumption, is considered of lower importance. However, some interviewees desire possibilities to analyse memory consumption, mainly to be used when the available resources are running low, i.e., before additional resources (ECUs) have to be added to the system.

4.3.4 Operating systems

In this section, we describe the issues involved in choosing operating system and the execution models used in the examined applications. We describe the main motivations to why these operating systems were chosen and the interviewees expressed experience of the used execution models. When investigating the type of technical considerations that has bearing on the choice of OS for the embedded applications, we discovered several non-technical considerations that are strong motivators to the choice of a specific OS, e.g., requirements on coordination to use a common OS at different departments of a company. These requirements do not directly reflect the technical need in development. The technical requirements are commonly considered later on. However, the requirements on simplicity, i.e., ease of use, is a motivator both when choosing OS and among available execution models The commercial operating systems that are used, or have been used, in the embedded applications by the investigated companies are Rubus [13], VxWorks [14], OSE [15], O'Tool [13], RTX [16] and WinCE [17]. In addition, one of the investigated companies develops their own operating systems, used in a majority of their applications. The main motivation for this is that their own operating systems are claimed to be simpler, more robust and have less run-time footprint (timing and memory overhead) than the commercial OSes. The interviewees' state that the main considerations when choosing a commercial operating system include:

- Cost (royalties, licenses).
- Availability of supported development tools related to the OS.
- The supported execution models in the OS, i.e., its suitability for the application domain.

- Coordination within a corporate group or subsidiaries to use a common OS.
- Recommendations originating from other companies evaluating the OS.
- The popularity of the OS, i.e., to what extent is the OS used by other companies.
- The OSs internal timing and memory overhead.
- Safety classification issues.

4.3.5 Execution models

Both time- and event-triggered execution models are used in all of the examined applications. The time-triggered model is commonly used for control functionality whereas the even-triggered model is used mainly for information handling for diagnostic reasons. The interviewees state that the choice of execution model in development is mainly dependent on: (i) Verification possibilities, both functional and temporal. (ii) Flexibility of adding new functionality. (iii) Required response-time on functionality. (iv) Simplicity of use in development.

4.3.6 Resource limitations

This section describes the current resource situation in the examined systems, as expressed by the interviewees. We investigated whether and to what degree, the amount of processing time, RAM, ROM and communication bandwidth, were considered constrained in the systems. As described in Section 4.3.3, many of the examined systems are intentionally over-dimensioned; hence, the interviewees did not consider any of the resources as being particularly constrained during software development. However, in case the systems would run out of resources, the interviewees' state that they would most probably consider installing additional hardware resources rather than redesigning the way the applications utilises the resources. This is however, said to be dependent on the urgency of system delivery. In extreme cases, functionality has been removed from the examined systems, when the available resources have been fully utilised.

4.3.7 Desired tool support

In this section, we present the interviewees expressed desire concerning support in development tools and their experiences of software components, i.e., component-based development [18]. The expressed wishes, concerning support in development tools, amplify the requirements on verification, safety and reliability aspects. The concise picture seems to be requirements on simulation and verification possibilities of applications on PCs. Moreover, an integrated possibility for model-based development with Matlab and Simulink together with automated code generation is another common desire expressed by the interviewees. The following is a list of desired tool support, as expressed by the interviewees. The desired support addresses both technical and process related issues. The interviewees would like to see:

- Simulation of the embedded applications on PCs.
- Replacing of text based user interfaces with graphical user interfaces.
- Support for model based development with possibilities to exchange information between tools from different vendors.
- Abstractions of graphical models i.e., visualisation of architectures at different levels and from different views.
- Automatic code generation e.g., from models to source code.
- Support for formal verification of source code.
- Support for execution time analysis.
- Possibilities to identify or trace the requirement specifications from the source code, and vice versa.

The current support in development tools varies at the companies. For example, one company has extensive support for simulation of embedded applications on a PC, whereas others do not have simulation possibilities at all. However, none of the examined companies has all of the listed support in their development tools.

4.3.8 Software components

Only one of the examined companies explicitly state that they use software components in the development of their applications. The company uses both in-house as well as third party developed components. The reasons to why the other investigated companies do not use software components are related to facts such as difficulties in understanding the concept of component-based development. Furthermore, issues such as modifiability of functionality are stated as a restricting factor for use of software components. However, all of the interviewees' state that the abstraction possibilities that components provide, is one of the main motivators of component based development, simply because it facilitates understanding and communication between developers.

4.4 Discussion - our observations

In this section, we address the main questions of the study and present our own observations and conclusions of the interviews.

Q1. What characterise the embedded applications?

The fact that more and more mechanical solutions are replaced with software, results in an increasing complexity both in size and in diversity. The applications are evolving and contain more heterogeneous functionality that before. In the future, this requires abilities to cope with (i) increasing data handling and (ii) increasing complexity in control functionality. It is common that applications contain a mix of hard and soft real-time tasks. We observed that a surprisingly small fraction (e.g., 25% at company A) of the requirements reflects need of hard real-time tasks. Still, the use of hard real-time tasks is very high (75% at company A). We believe that the high utilisation of hard tasks is mainly related to three reasons: (i) simplicity in development (ii) for verifications/reproducibility reasons (iii) tradition in development. The simplicity in development originates from years of evolving support in development tools that to large extents is intended for development of safety critical real-time systems. There is also a tradition in using hard real-time tasks for the majority of functionality, simply because developers tend to rely on designs from previous projects, instead of scrutinizing and considering the designs appropriateness for the diverging type of functionality found in today's and in future applications. Hence, the predicted increase in information intensity and diversity of functionality, require use of more suitable development models, i.e., models for diverging strategies that can handle both safety critical functionality as well as more flexible and resource efficient functionality in the same system.

Q2. What are the designers concern on application properties such as safety, maintainability, testability, reliability, portability and reusability?

The future classification of functionality in Safety Integrity Levels (SIL) implies that reliability, safety, analysability and testability will continue to be very important application properties in the future. Moreover, we believe that facilitating maintainability of the applications will be a more important activity to consider due to the increasing complexity, long product life cycles and demand on upgradeability of the applications. However, moving into the area of more maintainable systems, through, e.g., raising the level of abstraction and introducing reusable frameworks, introduces challenges since it must be done without compromising the systems safety or reliability.

Q3. How are the applications verified/analysed?

Functional behaviour is typically verified through testing on the target platform, whereas properties such as temporal behaviour are mainly verified with support of software analysis tools. Worst-case execution times are commonly estimated during development, and later on, verified through measurements on the target platform. The interviewees desire tools for verification of both functional and temporal behaviour of embedded applications on PCs. We believe that for the large fraction of future functionality, predictable and flexible execution models, where combinations of different analysis techniques that focus more on average case behaviour and quality of service rather than on worst-case behaviour, will be significant.

Q4. What are the considerations in choosing an OS, and execution model?

Politics and non-technical aspects are strong motivators in choosing OS. It is obvious that such issues could motivate the use of an OS that is more or less suitable to fulfil the technical issues in an application domain or specific needs within a corporate group. We believe that the increasing complexity in the examined application domain require more focus on technical issues, such as availability of novel tool support related to the OS and possibility to utilise more suitable execution models in the OS. For example, with increasing demand on safety classification such as SIL, the OS must be able to support the trade off between technical aspects such as verifiability and efficiency. For example, the small core of safety critical functionality should be allowed to use more resources if it must fulfil the SIL classification and be verifiable (testable and analysable), whereas the rest of the functionality (non-safety-critical) should utilise more resource efficient run-time mechanism to implement the functionality.

Q5. What resources are constrained in the systems, and to what degree?

Our investigation revealed that the computational resources are not considered as constrained during software development. We believe there are two possible reasons for this. (i) The investigated companies are already using resource efficient development methods (legacy methods), originating from times when all functionality was homogenously implemented. (ii) The systems are over-dimensioned at the same time as the developers put most effort in implementing complex functionality without having tool support to analyse resource consumption, e.g., memory usage. The increasing number of ECUs reveal that the computational resources are highly utilised from time to time, i.e., before addition of hardware. With an increase in diverging functionality, the current situation where a static schedule is used for the majority of all functionality, will either be intractable or overly resource demanding (ending up in new ECUs being added) in the future. Instead, the future development tools need to support an efficient and verifiable way to allocate resources, so that the developers either can: (i) Continue their efficient way of developing with efficient tool support adapted to the diverging functionality in the application domain, or (ii) Have novel tool support that allows them to begin developing systems using efficient and resource saving models. Some interviewees experience that the quality of software increases when developers do not have to worry about resource consumption. Hence, future support for resource efficient development needs to be automated to as large extents as possible.

Q6. What kind of tool support is needed in the development of future systems?

The view on future requirements is that safety critical functionality needs to be certifiable and the emphasis on less critical functionality will be on more efficient resource usage (e.g., average resource utilisation rather than worst case utilisation). This requires system integration tools with possibilities to take domain specific models that support efficient automatic code generation, reproducibility for the safety critical functionality and efficient resource usage for the rest. In addition, to cope with the increasing complexity developers need tools that lift the level of abstraction, i.e., tools that provide both different levels of abstraction as well as different views (e.g., temporal and functional) at each level of abstraction. It is imperative that the tools relieve some burden of developers (our study show that simplicity is a strong motivator in development) for example by letting synthesis tools provide details (such as assigning temporal attributes, priorities etc.) so that requirements are met.

Q7. What are the designers experiences of software components, i.e., component based development

There is an ongoing activity at one of the investigated companies concerning reusability of general type (application independent) software components. We believe that general components facilitate development and may increase the software quality since they are often adapted in several applications and being subject to extensive testing. However, to be resource efficient, or predictable for safety critical parts, these type of components need to be efficiently, and/or predictably, synthesised, i.e., become application specific in the run time system. Hence, the components should be general and execution model independent during development, and then mapped to an application specific run-time structure.

4.5 Conclusions

In this paper, we presented some requirements in development of industrial embedded systems in the vehicle domain. The requirements were collected by a number of interviews with ten senior designers at four companies in Sweden. Many of the investigated applications are developed using methods that are adequate for the (relatively small) parts that are safety critical. Less critical parts are adapted to fit into the framework of the critical parts. With the increasing size and complexity of software, this homogenous way of developing applications will, we believe, be inadequate. In the future software development strategies, methods and tools must be able to capture the different diverse requirements of the applications and trends in the application domains. Ranging from a small core part of the application that is safety critical to a larger part of the system focused on, for example, quality of service and average case behaviour. The characteristics of the examined systems and the predicted increase in information intensity and higher precision on control functionality, would allow for more suitable execution models, i.e., resource saving and quality enhancing, to be introduced (one company even expressed their interest in execution models addressing variable quality of service levels). A wide spectrum of different kind of tool support is desired in development of the applications. For example, tools for model-based development with simulation possibilities and automatic code generation are considered as highly desirable. Furthermore, the use of software components and CBSE in general, provides possibilities for architectural descriptions at a high level. The importance of architectural descriptions as means of communication between developers, i.e., not only as logical or structural system descriptions, implies that a strong motivator to use software components is their ability to serve as descriptive entities, i.e., not only as reusable entities.

4.6 Verification of the investigation results

According to Robson [9] there are no standardised means of assuring complete reliability in a study that use flexible design strategy. We did however follow recommendations in [9] to minimise threats to the reliability of the conducted study by:

- Studying and minimising possible sources of biases.
- Describing the application characteristics, properties etc. (Section 4.3) based on information from notes and tape recordings taken during the interviews.
- Interpreting the respondents answers at a group basis when necessary.
- Verifying the observations (Section 4.4 and Section 4.5), with the help of a senior designer with expertise in vehicular real-time systems.
- Verifying our observations (Section 4.4) with representatives from two of the participating companies.

Bibliography

[1] B. Graaf, M. Lormans, and H. Toetenel. Embedded software engineering: The state of the practice. *IEEE Software*, 20(6), 2003.

[2] B. Graaf, M. Lormans, and H. Toetenel. Software technologies for embedded systems: An industry inventory. In *4th International Conference on Product Focused Software Process Improvement*, Rovaniemi, Finland, 2002.

[3] P. Koopman. Embedded systems design issues(the rest of the story). In *Proceeding of the International Conference on Computer Design(ICCD)*, Austin, October 1996.

[4] A. Möller, J. Fröberg, and M. Nolin. Industrial requirements on component technologies for embedded systems. In *International Symposium on Component-based Software Engineering(CBSE7)*, Edingburgh, Scotland, May 2004.

[5] A. Möller, M. Åkerholm, J. Fröberg, and M. Nolin. Industrial grading of quality requirements for automotive software component technologies. In *Embedded Real-Time Systems Implementation Workshop in conjunction with the 26th IEEE International Real-Time Systems Symposium*, Miami, USA, December 2005.

[6] C. Norström, M. Gustafsson, K. Sandström, J. Mäki-Turja, and N-E. Bånkestad. Experiences from introducing state-of-the-art real-time techniques in the automotive industry. In *Eight Annual IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, Washington, US, April 2001.

[7] P.G. Paulin, C. Liem, M. Cornero, F. Nacabal, and G. Goossens. Embedded software in real-time signal processing systems: Application and architecture trends. In *Proceedings of the IEEE, Volume 85, Issue 3*, 1997.

[8] M. Åkerholm, J. Fredriksson, K. Sandström, and I. Crnkovic. Quality attribute support in a component technology for vehicular software. In *Fourth Conference on Software Engineering Research and Practice in Sweden*, Linköping, Sweden, October 2004.

[9] C. Robson. *Real World Research, 2nd edition*. Blackwell Publishing, 2002.

[10] M G.E. Peterson. User satisfaction surveys, what the engineer should know. In *Proceedings of the Ninth IEEE Symposium on Computer-Based Medical Systems*, June 1996.

[11] R. Yin. *Case Study Research, 3rd edition*. Sage Publications, 2003.

[12] International Electrotechnical Commission (IEC). *Functional safety and IEC 61508*. Geneva, Switzerland, May 2000.

[13] Arcticus systems. Web page, http://www.arcticus-systems.se.

[14] Wind River. Web page, http://www.windriver.com.

[15] Enea Embedded Technology. Web page, http://www.ose.com.

[16] Ardence. Web page, http://www.vci.com.

[17] Microsoft. Web page, http://msdn.microsoft.com/embedded/prevver/ce.net/.

[18] I. Crnkovic and M. Larsson. *Building Reliable Component-Based Software Systems*. Artech House, 2002.

Chapter 5

Paper B: Efficient Development of Real-Time Systems Using Hybrid Scheduling

Jukka Mäki-Turja, Kaj Hänninen, Mikael Nolin

In Proceedings of the International Conference on Embedded Systems and Applications, Las Vegas, USA, June, 2005.

Abstract

This paper will show how advanced embedded real-time systems, with functionality ranging from time-triggered control functionality to event-triggered user interaction, can be made more efficient. Efficient with respect to development effort as well as run-time resource utilization. This is achieved by using a hybrid, static and dynamic, scheduling strategy. The approach is applicable even for hard real-time systems since tight response time guarantees can be given by the response time analysis method for tasks with offsets.

An industrial case study will demonstrate how this approach enables more efficient use of computational resources, resulting in a cheaper or more competitive product since more functionality can be fitted into legacy, resource constrained, hardware.

5.1 Introduction

As the complexity of embedded real-time systems keeps growing, both by increases in size and in diversity, the developers are faced with the increasing challenge of modelling, analyzing, implementing and testing both the functional as well as the temporal behavior of these systems. This paper will present ways to simplify some of that complexity by introducing methods to verify the temporal correctness for a larger class of such systems.

Traditionally, one design parameter has been what execution model to choose. Two common and widespread execution models are the static and dynamic execution models:

- **Static scheduling**, where a schedule is produced off-line. The schedule contains all scheduling decisions, such as when to execute each task or to send each message. During run-time a simple dispatcher dispatches tasks according to the schedule. Static scheduling is sometimes referred to as time-triggered scheduling.
- **Dynamic scheduling**, where scheduling decisions are made on-line by a run-time scheduler. Typically some task attribute (such as priority or deadline) is used by the scheduler to decide what task to execute. The scheduler implements some queueing discipline, such as fixed priority scheduling or earliest deadline first. Dynamic scheduling is sometimes referred to as event-triggered scheduling.

Since both models have their pros and cons, the design decision of which one to use is not simple. A few trade-offs when choosing execution model are:

- **Overhead** – Since all scheduling and synchronization decision are made off-line in the static approach, the run-time overhead for scheduling is kept low. In dynamic scheduling these decisions are made on-line, often resulting in a larger overhead.
- **Responsiveness** – Statically scheduled systems are inflexible and have therefore limited possibility in responding to dynamic events, resulting in poor responsiveness. Dynamically scheduled systems, on the other hand, handles dynamic events naturally and can provide high degree of responsiveness.
- **Resource usage** – In order to provide some degree of responsiveness for dynamic events in the environment, statically scheduled systems tend to waste resources on redundant polling, whereas event-triggered dynamic schedulers only handle the actual events, enabling better service to soft or non-real time functionality when events do not occur at their maximum rate.
- **Overload** – In static scheduling the effects of overload are highly predictable.

The exact capacity, e.g. in terms of number of inputs handled, is known and the effect of lost events, e.g. due to slow polling, can be predicted. In dynamic scheduling, no natural overload control is inherent. Instead, ad-hoc mechanisms are used to prevent, e.g., faulty sensors from flooding the systems with interrupts. A dynamically scheduled system which becomes overloaded is unpredictable, it is often difficult to assess which buffer will overflow and thus which tasks will miss their deadlines.

- **Determinism** – A statically scheduled system is highly deterministic, it executes according to the pre-defined schedule each time. A dynamically scheduled system, on the other hand, may exhibit different behavior each time the system is run, due to, e.g., race conditions on shared resources. This has a major impact on reproducibility, and thus also on the functional testability, of the system. Determinism also simplifies the verification process which is a major part when certifying safety critical applications.
- **Complex constraints** – Statically scheduled systems can handle more complicated inter-task relation constraints. For example, control systems, where control performance is important, need to have small (input and/or output) jitter, which is easier to accommodate in a static scheduler than with simpler dynamic scheduling parameters.
- **Adding new functionality** – Once a static schedule has been constructed it can be very hard to add new functionality in the system, a completely new schedule has to be constructed. For a dynamically scheduled system, new functions can be added with a minimum of impact on other parts of the system.

For further discussions on these trade-offs see [1] which advocates cyclic scheduling), and [2] which advocates dynamic, fixed priority, scheduling.

As can be seen, both approaches have their virtues and one often wishes to have both approaches available when developing embedded real-time applications. This desire is clearly illustrated by the last few years of development in the area of field busses for automotive applications. The Controller Area Network (CAN) [3] has been predominant in the automotive industry. CAN provides dynamic scheduling (using fixed priority scheduling). However, the automotive industry felt a need for a more dependable and predictable bus architecture. So when Kopetz brought attention to his Time Triggered Protocol (TTP) [4], which provides static scheduling, many automotive manufacturers and their sub-contractors embraced the new technology. It was soon recognized that TTP was a bit *too* static. Hence, a consortium of automotive manufacturers and sub-contractors started the development of FlexRay [5], which provides both static and dynamic scheduling. Also, on the operating-system side, products that support both static and dynamic scheduling have emerged. For instance, Arcticus Systems' operating system Rubus [6], and the open source real-time operating system Asterix [7]. In fact, most priority driven operating systems can implement hybrid static and dynamic scheduling by letting a dispatcher (a time-table) execute at highest priority.

Thus, we see that the need to combine static and dynamic scheduling have led to some practical solutions available today. However, one problem with systems that tries to combine static and dynamic scheduling is that they often consider the dynamic part as non real-time, e.g. [6, 5]. That is, dynamic scheduled tasks/messages are not given any response-time guarantees, only best-effort service is provided. However, in order to fully utilize the potential of combining static and dynamic scheduling in hard real-time systems, both the dynamic and the static parts need to be able to provide response-time guarantees. A recent study of industrial needs recognizes that one of the key issues for embedded systems is analyzability [8].

This paper presents a method to model hybrid, statically and dynamically, scheduled systems with the task model with offsets [9]. With this model, and the corresponding response time analysis, tight response time guarantees can be given also for dynamically scheduled tasks. The modelled system can be realized with commercially available operating systems support. Furthermore, in a case study we show how a legacy system at Volvo Construction Equipment could benefit from this approach by migrating functionality from the resource demanding statically scheduled part to the dynamically scheduled part, freeing system resources while still fulfilling original temporal constraints.

Paper Outline: Next, Section 5.2 describes the type of systems studied in this paper. Section 5.3 shows how these systems can be modelled using the task model with offsets. Section 5.4 discusses related work. Section 5.5 illustrates, through a case study, how this approach can be applied to a legacy system, migrating functions from a static schedule, freeing system resources. Finally, Section 5.6 presents our conclusions.

5.2 System description

In this paper, we address the issue of providing tight response-time guarantees to dynamically scheduled tasks running "in the background" of a static schedule. The system model contains:

- **Interrupts**. There may be multiple interrupt levels, i.e., an interrupt may be preempted by higher level interrupts.

- **A static cyclic schedule**.
  - ○A set of periodic static tasks (functions) are scheduled in the schedule. Each task has a known worst case execution time (WCET).
  - ○The schedule has a length (a duration) that is equal to the LCM (least common multiple) of all statically scheduled function periods. The schedule is constructed off-line by a scheduling tool.
  - ○Each function is scheduled at an offset relative to the start of the schedule. This is also referred to as a function's *release time*.
  - ○The static cyclic scheduler activates each function in the schedule at its release time. When the whole schedule has been executed the schedule is restarted from the beginning.

  Interrupts may preempt the execution of statically scheduled functions.
- **A set dynamically dispatched tasks**. We call each such task a *dynamic task*. These tasks executes in the time slots available between interrupts and statically scheduled functions. Dynamic tasks are scheduled by a fixed priority preemptive scheduler. They are assumed to be periodic or, at least, to have a known minimum time between two invocations.

We assume that a static cyclic schedule has been constructed prior to the analysis of dynamic tasks. Furthermore, we assume that the schedule is valid even if its functions are preempted by interrupts. How a scheduler can generate a feasible schedule, with interfering interrupts, is described in [10].

5.2.1 Example system

Fig. 5.1 shows a static cyclic schedule of length 20, with 4 functions released at times 0, 5, 10 and 15, with WCETs 4, 1, 1 and 3 respectively.

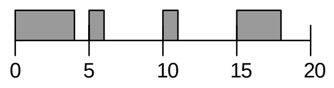

Figure 5.1: Example of static cyclic schedule

In Fig. 5.2 we see an example execution scenario when executing the schedule from Fig. 5.1, with one interfering interrupt source and one dynamically scheduled task (two instances of that task are activated). We make the observation that both interrupts and the static schedule act like higher priority tasks from the dynamic tasks' point of view.

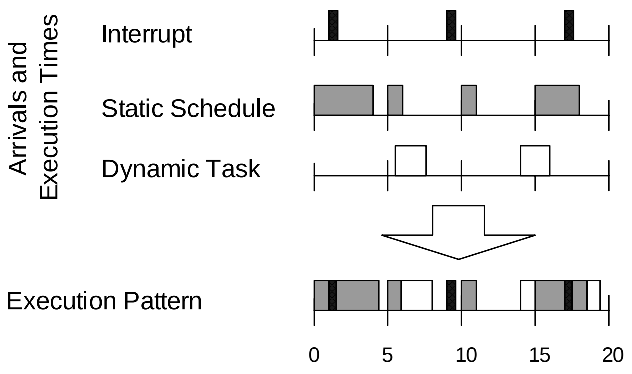

Figure 5.2: Example execution scenario

One of the main objectives of this paper is to enable response-time calculations for dynamic tasks. The goal is to model static schedules (and interrupts) so as to incur as little interference on dynamic tasks execution as possible. Thus, modelling both functions' WCETs as well as their release times as accurately as possible.

5.3 Modelling the system

Classical response-time analysis (see e.g. [11, 12, 13]), assumes that a critical instant[1] occurs when all tasks are released simultaneously. Using this model, the static schedule described in Section 5.2, can be modelled as 4 tasks. These tasks would have a period of 20 and WCETs of 4, 1, 1, and 3 respectively. However, this approach is overly pessimistic since it assumes that all four static tasks can be released for execution at the same time. In our example, assuming no interrupt interference, a dynamic task with a WCET of 1, would have a response time of 10 (4+1+1+3+1). However, looking at Fig. 5.1 one can see that the actual worst possible response-time is 5 (if the dynamic tasks coincides with the static function scheduled at time 0).

In static schedules it is impossible for all static tasks to start at the same time. The task model with offset introduced by [14, 15] is able to capture the

[1]Point in time, where the task under analysis is released for execution, resulting in the longest possible response-time.

time separation in static schedules, and thus reduce the pessimism. In [9] we further reduced the pessimism in the corresponding response time formulae.

5.3.1 Task model with offsets

The task set, Γ, in [9] consists of a set of k transactions, $\Gamma_1, \ldots, \Gamma_k$. Each transaction Γ_i is activated by a periodic sequence of events with period T_i. A transaction Γ_i, contains $|\Gamma_i|$ number of tasks, and each task is activated when a relative time, *offset*, elapses after the arrival of the event.

τ_{ij} is used to denote a task. The first subscript denotes which transaction the task belongs to, and the second subscript denotes the number of the task within that transaction. A task τ_{ij} is defined by a worst case execution time (C_{ij}), an offset (O_{ij}), a deadline (D_{ij}), maximum jitter (J_{ij}), maximum blocking from lower priority tasks (B_{ij}), and a priority (P_{ij}). The task set Γ is formally expressed as follows:

$$\begin{aligned}
\Gamma &:= \{\Gamma_1, \ldots, \Gamma_k\} \\
\Gamma_i &:= \langle \{\tau_{i1}, \ldots, \tau_{i|\Gamma_i|}\}, T_i \rangle \\
\tau_{ij} &:= \langle C_{ij}, O_{ij}, D_{ij}, J_{ij}, B_{ij}, P_{ij} \rangle
\end{aligned}$$

There are no restrictions placed on offset, deadline or jitter. The maximum blocking time for a task, τ_{ij}, is the maximum time it has to wait for a resource which is locked by a lower priority task. In order to calculate the blocking time for a task, usually, a resource locking protocol like priority ceiling or immediate inheritance is needed. Algorithms to calculate blocking times for different resource locking protocols are presented in [16]. Priorities can be assigned with any method (e.g. rate monotonic, deadline monotonic, or user defined priorities). One must assume that the load of the task set is less than 100%.[2]

Parameters for an example transaction (Γ_i) with two tasks (τ_{i1} and τ_{i2}) is depicted in Fig. 5.3. The offset denotes the earliest possible release time of a task relative to the start of its transaction and jitter (illustrated by the shaded region) denotes maximum possible variability in the actual release of a task. The upward arrows denotes earliest possible release of a task and the height of the arrow corresponds to the amount of execution released. The end of the shaded region represents the latest possible release of a task.

[2]This can easily be tested, and if not fulfilled some response-times may be infinite; rendering the task set unschedulable.

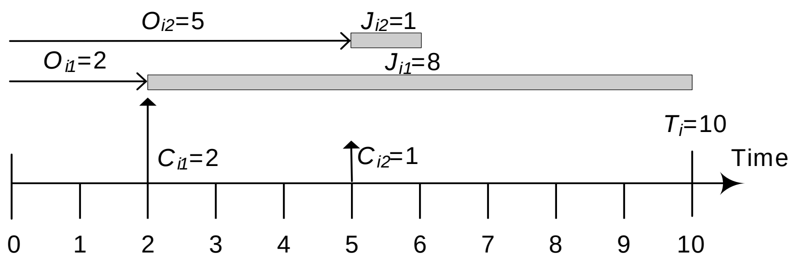

Figure 5.3: Example transaction

5.3.2 System model

The system in Section 5.2 can be modelled, and dynamic tasks subsequently analyzed for response times, with the above task model as follows (subscripts $i, s,$ and d denote a generic interrupt, static, and dynamic transaction respectively):

- **Each interrupt** will be modelled as a transaction, Γ_i, containing one single task (i.e., $|\Gamma_i| = 1$) with T_i set to minimum inter-arrival time of the corresponding interrupt. These interrupt tasks will have the highest priorities in the system. If there are several interrupt levels, priorities are assigned accordingly, i.e., highest priority to highest interrupt level.
- **The static schedule** is modelled as one transaction, Γ_s, where each release time in the schedule is modelled as one task, τ_{sj}, where the offset ,O_{sj}, is set to the corresponding release time. The worst case execution time, C_{sj}, is set to the corresponding functions WCET. The priority, one suffices, for static tasks must be lower than for any interrupt, but higher than those for dynamic tasks.

  Our example schedule of Fig. 5.1 will be modelled as a transaction ($T_s = 20$) with 4 tasks, with offsets 0, 5, 10, 15 and worst case execution time of 4, 1, 1, 3 respectively.
- **Dynamic tasks** will have the most variability on how they are modelled. In the simplest case they are modelled exactly the same way as interrupts but with lower priorities. This situation corresponds to simple periodic (or sporadic) dynamic tasks with no jitter, no time separation (offsets), and no blocking. However depending on the nature of the dynamic tasks their corresponding transaction can be extended by:

  ○jitter if there is variability in their periodicity,

- ◦by blocking if they share resources and providing the run-time system supports an analyzable resource sharing protocol, and
- ◦offsets if there are temporal dependencies, such as precedence, among dynamic tasks.

Note that dynamic tasks cannot communicate with static tasks, via locked resources, since they must not affect their temporal behavior. However, there exist methods to communicate between these two systems that will not affect the temporal behavior of static tasks, see e.g. [17].

Assuming the dynamic task of Fig. 5.2 is a sporadic task with minimum inter-arrival time of 10 time units and a release jitter of 3 time units, it is modelled as a transaction with $T_d = 10$ containing one task with $J_{dj} = 3$. The execution time is 2 and since it is the lowest priority task the blocking is zero ($C_{dj} = 2$ and $B_{dj} = 0$).

The formulae to calculate the response times rely on a relaxed critical instant assumption stating that only one task out of every transaction has to coincide with the critical instant. The complete formulae can be found in [9], and would, for our example system of Fig. 5.2, result in a response time of 5 time units for a dynamic task with $C_{dj} = 1$, assuming no interrupt interference.

Since all type of tasks, interrupt, static, and dynamic, can be analyzed for responsiveness, the inability of providing response time guarantees will no longer be a basis for rejecting an execution model for a function, thus making hybrid static and dynamic scheduling suitable even for hard real-time systems.

5.4 Related work

There has been number of research projects addressing the issue of combining several execution models [18, 19, 20]. These provide reservation-based guarantees where task characteristics are not fully known in advance. Furthermore, no commercially available real-time operating system support exist for them. Our approach is to model existing systems, supported by commercial RTOSes, where task attributes are fully known at design time. However, [21] aims at modelling real situations through hierarchically modelling different schedulers. They cover preemptive and non-preemptive priority schedulers and do not model static schedulers. In fact, the work presented in this paper could extend their more general framework with the ability to model also static schedulers.

5.5 Case study

A case study [22] conducted at Volvo Construction Equipment (VCE) [23], with the objective of finding a way to use available resources in a more efficient way has studied the design trade-offs between static and dynamic scheduling.

VCE has a tradition in statically scheduled systems. This is mainly due to the safety critical nature of their control systems in their heavy machinery, e.g., articulated haulers, trucks, wheel loaders and excavators. Rubus OS by Arcticus [6], used by VCE, has run-time support for the system model described in Section 5.2.

Currently at VCE, all safety critical functionality is implemented in the static part and only soft real-time or non real-time activity resides in the dynamic part. In recent interviews (in an ongoing research project) they state that about 20-25% of their applications are considered safety critical, mainly residing in transmission and engine control. However, some operational modes, have static schedule utilization as high as 74%.

The demand on more functionality in next generation machinery is growing. However, the static schedule is getting close to full utilization, leaving little or no room for new functionality. This can either be addressed with new and more expensive hardware or to find a better way of utilizing the current hardware resources.

Demand on responsiveness (i.e. deadlines) for functionality in the static part ranges from a few milliseconds up to several seconds. This could potentially result in very large schedules (with corresponding high memory consumption). VCE's solution to this has been to fix the schedule length at 100ms, which result in waste of computing resources due to redundant polling for any function with a responsiveness demand higher than 100ms (even functions with responsiveness demand within 100ms but associated with events that occur seldom will in this case waste computing resources). A solution that could get rid of this redundant polling, while still guaranteeing the responsiveness and without increasing the schedule length, would be highly desirable.

5.5.1 An example system

Here we will present an example system that can be viewed as a simplified version of one of the systems constructed by VCE. A complete system would consists of several hundreds of tasks [22] and would be too complex to present in this paper. We will show how functions currently residing in the static part can be moved to the dynamic part and, by using the response-time analysis of [9], still guarantee that the function deadlines will be met. Type of functionality that could be moved, according to [22], consists of events that by nature are event-triggered, visual interaction with driver, and logging of operational statistics. Another example of functionality that may be moved to the dynamic part is control functionality that is not part of sampling or actuation. Control performance is often sensitive to jitter in sampling and actuation and therefore often placed in a static schedule [24]. However, the control calculation and updating of control state do not have these strict requirements on jitter and their responsiveness requirement is only restricted by the corresponding output action and sampling in the next period respectively. Therefore control and updating control state functionality could be moved to the dynamic part.

Task i	T_i	C_i	D_i	U^{100}	U^T
A	10	2	10	20%	20%
B	20	2	5	10%	10%
C	50	1	2	2%	2%
D	50	6	50	12%	12%
E	100	8	100	8%	8%
F	2000	7	100	7%	0.35%
G	2000	8	100	8%	0.4%
H	2000	8	2000	8%	0.4%

Table 5.1: The set of tasks in the Static system

For our example, the task specification in table 5.1 will be used. (For simplicity we will in this example ignore interrupt interference.) Tasks F and G handle events that may occur once every 2000ms, and with a response time requirement of 100ms. Placing tasks F and G in a static schedule, means that they would have to be polled at the rate of their deadline (100ms) instead of their period (2000ms) (since we do not know exactly when the events are going to occur). Task H, however, could be polled at the rate of its period (2000ms), however, the resulting schedule would become too large and memory consuming (it would have to extend for 2000ms and thus consume over 20kb of ROM). Setting the schedule length to 100ms would be adequate for all tasks except task H. Hence, the schedule length is set to 100ms, and a resulting schedule can be seen in Fig. 5.4 on the facing page.

In table 5.1, U^{100} represents the task utilization when scheduled in a static schedule with a period of 100ms, and U^T represents the utilization when tasks are scheduled with their period.

The total utilization of the static schedule is 75%. Adding new functional-

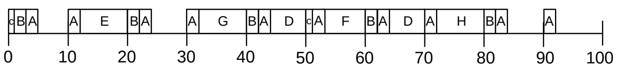

Figure 5.4: Static schedule for table 5.1 task set

ity, requiring some kind of temporal guarantee, to this system can be difficult, there are not many free time-slots in the schedule, especially if there has to be room also for interrupts and non-real-time functionality.

Improving the system

However if tasks F, G, and H could be made event triggered, by placing them in the dynamic part of the Rubus OS, some resources could be freed. The resulting static schedule can be seen in Fig. 5.5. The utilization for the static schedule now becomes 52%. The utilization for the three dynamic tasks are 1,15%, resulting in a total utilization of just above 53%. Thus, by moving these three tasks from the static schedule we free nearly 22%[3] of the CPU resources.

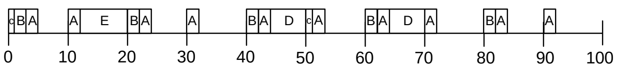

Figure 5.5: Schedule without tasks F, G and H

Now, it remains to see whether the three tasks will meet their deadlines when running as dynamic tasks. To be able to calculate response times for tasks F, G, and H we model the static schedule as a transaction with $T_s = 100$. WCETs and offsets are set as follows:

$$C_{sj} = (5,\ 10,\ 4,\ \ 2,\ 10,\ 3,\ 10,\ \ 2,\ 4,\ 2)$$
$$O_{sj} = (0,\ 10,\ 20,\ 30,\ 40,\ 50,\ 60,\ 70,\ 80,\ 90)$$

Assuming that F, G, and H have priorities high, medium, and low respectively, we can calculate the response times for the three tasks according to [9].

[3]Increase in overhead for tasks F, G, and H as dynamic tasks will be marginal, hence not considered here.

And the result is:

$$R_F = 26 \quad R_G = 44 \quad R_H = 64$$

We see that all three tasks will meet their deadlines of table 5.1. In fact, their responsiveness is considerably increased compared to being statically scheduled every 100ms. It could be mentioned that by removing tasks F, G and H from the schedule we have enabled shorter response times for other dynamic tasks, that might have existed in the system, as well. The schedule in Fig. 5.4 has a longest busy period of 54ms (between 30–84), whereas the new schedule in Fig. 5.5 has a longest busy period of 14ms (between 10–24). Since any dynamic task (in the worst case) will have to wait for the longest busy period, we now have significantly reduced that time.

With the approach presented in this paper the static schedule could be kept small (with respect to memory consumption as well as utilization). By modelling the static schedule as one transaction, response time analysis for task with offsets can be used to evaluate timeliness for the dynamic part.

Our solution reduce utilization by moving functionality, previously polled excessively, from the static schedule to the dynamic part. Our method also gives a possibility to shrink the static schedule since functions with long periods can be moved from the static schedule. It should be mentioned however, that all tasks in the static schedule share a common stack, whereas moving tasks from the schedule to the dynamic part may require them to have separate stacks, hence increasing the memory consumption for dynamic tasks. However, using a resource locking protocol such as the immediate inheritance allows also dynamic tasks to share a single stack [16, 25].

The possibility to selectively migrate functions from static scheduled legacy systems to dynamic scheduled systems will substantially facilitate for companies to gradually move into the area of dynamic scheduling, and thus, in the long run, help companies to use cheaper hardware for, or fit more functions into, their products. Also the development process becomes easier because event triggered functionality does not have to be force-fitted into a static model.

5.6 Conclusions

As stated in [8] analyzability is one of the major concern for embedded systems development. We have in this paper shown how a hybrid, static and dynamic, scheduling model can be modeled and dynamic tasks analyzed for responsiveness. The type of system presented can be realized by commercially available

OS support, e.g., Rubus OS by Arcticus [6]. In fact, any fixed priority OS complemented with an external static scheduler can implement this type of system with the static schedule as a task at highest priority.

A hybrid, static and dynamic, scheduling model simplifies the design tradeoffs of which scheduling model to choose. Appropriate scheduling model can be chosen on function level instead of system level. Since temporal guarantees can be provided, this approach will also be applicable for hard real-time systems. Choosing the most appropriate model for each function, instead of force-fitting it to an overall model, not only simplifies the design choices but also gives the possibility to save system resources and improve responsiveness. This is demonstrated in a case study [22] at Volvo Construction Equipment using the commercial real-time operating system Rubus by Arcticus [6].

Bibliography

[1] J. Xu and D.L. Parnas. Priority Scheduling Versus Pre-Run-Time Scheduling. *The Journal of Real-Time Systems*, 18(1):7–23, January 2000.

[2] C.D. Locke. Software architecture for hard real-time applications - cyclic executives vs. fixed priority executives. *The Journal of Real-Time Systems*, 4:37–53, 1992.

[3] Road Vehicles – Interchange of Digital Information – Controller Area Network (CAN) for High Speed Communications, February 1992. ISO/DIS 11898.

[4] H. Kopetz and G. Grünsteidl. TTP – A Protocol for Fault-Tolerant Real-Time Systems. *IEEE Computer*, pages 14–23, January 1994.

[5] FlexRay Home Page. http://www.flexray-group.org/.

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

[7] The Asterix Real-Time Kernel. http://www.mrtc.mdh.se/projects/asterix/.

[8] Anders Möller, Joakim Fröberg, and Mikael Nolin. Industrial Requirements on Component Technologies for Embedded Systems. In *7th International Symposium on Component-based Software Engineering (CBSE7)*. IEEE Computer Society, May 2004.

[9] Jukka Mäki-Turja and Mikael Nolin. Tighter Response-Times for Tasks with Offsets. In *Proc. of the 10^{th} International conference on Real-Time Computing Systems and Applications (RTCSA'04)*, August 2004.

[10] Kristian Sandström, Christer Eriksson, and Gerhard Fohler. Handling interrupts with static scheduling in an automotive vehicle control system. In *5th International Workshop on Real-Time Computing Systems and Applications (RTCSA '98)*, pages 158–165, Hiroshima, Japan, October 1998. IEEE Computer Society.

[11] N.C. Audsley, A. Burns, R.I. Davis, K. Tindell, and A.J. Wellings. Fixed Priority Pre-Emptive Scheduling: An Historical Perspective. *Real-Time Systems*, 8(2/3):173–198, 1995.

[12] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*. Addison-Wesley, second edition, 1996. ISBN 0-201-40365-X.

[13] M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390–395, 1986.

[14] J.C. Palencia Gutierrez and M. Gonzalez Harbour. Schedulability Analysis for Tasks with Static and Dynamic Offsets. In *Proc. 19th IEEE Real-Time Systems Symposium (RTSS)*, December 1998.

[15] K. Tindell. Using offset information to analyse static priority preemptively scheduled task sets. Technical Report YCS-182, Dept. of Computer Science, University of York, England, 1992.

[16] G.C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic Publishers, 1997. ISBN 0-7923-9994-3.

[17] Dag Nyström, Mikael Nolin, Aleksandra Tesanovic, Christer Norström, and Jörgen Hansson. Pessimistic Concurrency-Control and Versioning to Support Database Pointers in Real-Time Databases. In *Proc. of the 16th Euromicro Conference on Real-Time Systems*, June 2004.

[18] J. Regher and J.A. Stankovic. HLS: A framework for composing soft real-time schedulers. In *Proc. 22th IEEE Real-Time Systems Symposium (RTSS)*. IEEE Computer Society, December 2001.

[19] S. Saewong, R. Rajkumar, J.P. Lehoczky, and M.H. Klein. Analysis of hierarchical fixed priority scheduling. In *Proc. of the 14th Euromicro Conference on Real-Time Systems*. IEEE Computer Society, June 2002.

[20] Scott Brandt, Scott Banachowski, Caixue Lin, and Timothy Bisson. Dynamic Integrated Scheduling of Hard Real-Time, Soft Real-Time, and

Non-Real-Time Processes. In *Proc. 24th IEEE Real-Time Systems Symposium (RTSS)*. IEEE Computer Society, December 2003.

[21] J. Regher, A. Reid, K. Webb, M. Parker, and J. Lepreau. Evolving real-time systems using hierarchical scheduling and concurrency analysis. In *Proc. 24th IEEE Real-Time Systems Symposium (RTSS)*. IEEE Computer Society, December 2003.

[22] T. Riutta and K. Hänninen. Optimal Design. Master's thesis, Mälardalens Högskola, Dept of Computer Science and Engineering, 2003.

[23] Volvo Construction Equipment. http://www.volvoce.com.

[24] A. Cervin. Improved scheduling of control tasks. In *Proc. of the 11th Euromicro Workshop of Real-Time Systems*, pages 4 – 10, June 1999.

[25] Northern Real-Time Applications. SSX5 True RTOS, 1999.

Chapter 6

Paper C: Efficient Event-Triggered Tasks in an RTOS

Kaj Hänninen, John Lundbäck, Kurt-Lennart Lundbäck, Jukka Mäki-Turja, Mikael Nolin
In Proceedings of the International Conference on Embedded Systems and Applications, Las Vegas, USA, June, 2005.

Abstract

In this paper, we add predictable and resource efficient event-triggered tasks in an RTOS. This is done by introducing an execution model suitable for example control software and component-based software. The execution model, denoted single-shot execution (SSX), can be realized with very simple and resource efficient run-time mechanisms and is highly predictable, hence suitable for use in resource constrained real-time systems. In an evaluation, we show that significant memory reductions can be obtained by using the SSX model.

6.1 Introduction

When designing software for embedded systems, resource consumption is often a major concern. Software consumes resources primarily in two domains: the time domain (execution time), and the memory domain (e.g., RAM and flash memory). For systems without real-time requirements, the resource consumption in the time domain may be of less importance. However, most embedded systems are either used to control or monitor some physical process, or used interactively by a human operator. In both of these cases, it is often required that the system responds within fixed time limits. Hence, methods for development of embedded systems need to allow design of both memory and time efficient systems. Moreover, predictable use of the resources are required. Predictions of the amount of resources needed to execute the system are used to dimension the system resources (e.g., selecting CPU and amount of memory). The current trend in development of embedded systems is towards using high-level design tools with a model-based approach. Models are described in tools like Rational Rose, Rhapsody, Simulink, etc. From these models, whole applications or application templates are generated. However, this system generation seldom considers resource consumption. The resulting systems become overly resource consuming and even worse; they exhibit unpredictable resource consumption at run-time. In this paper, we describe and evaluate the integration of a resource efficient and predictable execution model, denoted single shot execution model (SSX), in a commercial real-time operating system. The execution model facilitates stack sharing to reduce memory consumption and priority scheduling to allow timing predictions. The paper is organized as follows. In section 6.2, we describe the properties of the SSX model and the prerequisites needed to utilize the model. In section 6.3, we describe our target platform, the Rubus RTOS. In section 6.4, we describe the integration of SSX in Rubus and in section 6.5, we evaluate the stack usage under the SSX model in different execution scenarios. In section 6.6, we conclude the integration of SSX in Rubus.

6.2 The single shot execution model (SSX)

Throughout the years, research in real-time scheduling and real-time operating systems has resulted in a vast number of different execution models, e.g.,[1][2][3][4][5], one of them being the single shot execution model in which tasks are considered to terminate at the end of each invocation, i.e., execute to completion (as opposed to indefinitely looping tasks). Baker [1] and Davis et al. [2] shows that the single shot execution model, with an immediate priority ceiling protocol, enables possibilities for efficient resource usage by stack sharing among several tasks. Stack sharing in the SSX model is feasible because higher priority tasks are allowed to pre-empt lower priority tasks and execute to completion (i.e., terminate) before lower priority tasks are allowed to resume their execution. However, the fact that a task must execute to completion (terminate) before any lower priority task is allowed to execute, puts some restrictions on suspensions of tasks in the SSX model:

- To guarantee correct stack access, self-scheduling of SSX tasks, i.e., calling timed sleep or delay functions may not be used in the application code of SSX tasks.

- Task synchronization should be done using the Immediate Priority Ceiling Protocol (IPCP). This ensures that a task will never be allowed to start executing, before it is guaranteed to have access to all resources it needs. Hence, calls for accessing shared resources, such as semaphores, will never result in blocking due to locked resources. Any possible blocking will occur before the task is allowed to start execute.

However, these design restriction also facilitate predictability since the administration of the tasks is left entirely to the operating system. Moreover, it is known that the immediate priority ceiling protocol is deadlock free and exhibits an upper bound on blocking times for tasks sharing resources. This implies that an analysis technique such as response-time analysis [6] enables analysis of temporal properties of an SSX system. The SSX model is conceptually very simple, at run-time a task can be in one of three states: terminated, ready, or executing. The main difference, from a developers view, is that a conventional RTOS often uses so called self-scheduled tasks. This means that a task is activated once, typically at system start-up, and eventually, after some possible initialization code, ends up in an infinite loop where it self-schedules itself, e.g., using delay calls. An SSX task, on the other hand, when activated by the OS, executes with no delay calls, and terminates upon completion. This means that such tasks have to be re-scheduled by the OS in order to provide a continuous service. Figure 6.1 illustrates the structural difference between a conventional task and an SSX task.

In this paper, we present an integration of the SSX model in the Rubus RTOS. We also present a quantitative evaluation of stack usage under different execution scenarios.

```
taskEntryFunc(){          taskEntryFunc(){
 while(1){
  //Task code               //Task code
  sleep(time)             }
 }
}
```

Figure 6.1: Looping task (left). Typical SSX task (right)

6.3 The Rubus operating system

Rubus is a real-time operating system developed by Arcticus Systems [7]. Rubus is targeted towards systems that typically require handling of both safety critical functions as well as less critical functions. The emphasis of Rubus is placed on satisfying reliability, safety and temporal verification of applications. It can be seen as a hybrid operating system in the sense that it supports both statically and dynamically scheduled tasks. The key features of Rubus RTOS are:

- Guaranteed real-time service for safety critical applications
- Best-effort service for non-safety critical applications
- Support for time- and even-triggered execution of tasks
- Support for component based applications

Rubus consist of three separate kernels (Figure 6.2). Each kernel supports a specific type of execution model.

The *Red kernel* supports time driven execution of static scheduled 'Red tasks', mainly to be utilized for applications with fixed hard real-time requirements. The static schedule is created off-line by the Rubus Configuration Compiler. Synchronizations of shared resources are handled by time separation in the static schedule. All tasks executed under the Red kernel share a common stack. A Red task is implemented by a C function, and the task is completed when the function returns. The *Blue kernel* administrates event driven execution of dynamic scheduled 'Blue tasks', mainly intended for applications having soft real-time requirements. Task handled by the Blue kernel are scheduled on-line by a fixed priority pre-emptive scheduler. Synchronizations among Blue tasks are managed by a Priority Ceiling Protocol (PCP)[8].

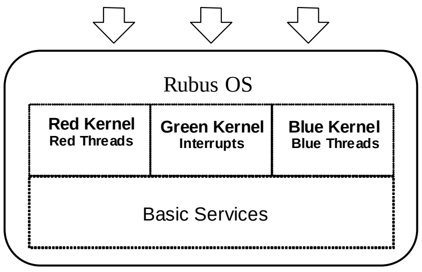

Figure 6.2: Rubus RTOS architecture

As opposed to the Red execution model, the Blue execution model does not support stack sharing among Blue tasks. Blue tasks are commonly used as indefinitely looping tasks (see Figure 6.1) periodically reactivated by system calls, e.g., blueSleep, that suspends the execution of Blue tasks for a specified time interval. The *Green kernel* handles external interrupts. The 'Green tasks' are scheduled on-line with a priority based scheduling algorithm dependent of the application hardware, i.e., microprocessor. The Rubus off-line scheduler is guaranteed to generate a static schedule (see Red kernel above) with sufficient slack available to handle interrupts [9]. When a Green task is executed, it may utilize the stack of the currently active Red or Blue task, implying that the active task may need to supply stack space for interrupt handling. Dispatch priorities of the tasks executing under the different kernels are illustrated in Figure 6.3. Tasks managed by the Green kernel have highest priority, and tasks managed by the Blue kernel have lowest priority.

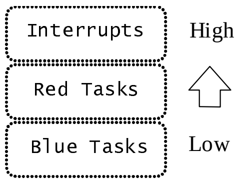

Figure 6.3: Task priorities in Rubus

Rubus supports the possibility to utilize software components for application development. The computational part of the supported software components is realized either by a Green, Red or by a Blue task.

6.4 Integration of SSX in Rubus

Introducing a new execution model in an operating system for resource constrained embedded real-time systems, require careful design to minimize the overhead of the new model and effects (temporal and spatial) on existing models. On one hand, we could minimize the memory overhead imposed by the new execution model, by sharing administrative code in the kernel between the existing execution models and the new execution model. In doing so, we would impose additional timing overhead on the existing models wherever a kernel needs to be able to separate the different models, e.g., at sorting, queuing and error handling etc. On the other hand, we could avoid imposing timing effects on the existing models by separating the models, i.e., modularize, and allow the kernel to administrate the new SSX model in isolation from the existing execution models. This approach would increase the number of administrative functions, thus requiring more memory. In this implementation, we choose to share administrative OS code between the SSX model and the Blue model since the timing overhead imposed by the SSX model, on the Blue model, is very low. A new execution model may be introduced to a system by changing the current scheduling policy or existing task model. In our case, we retain the same scheduling policy for the SSX model as for the Blue model (fixed priority scheduling). However, a new task model is introduced to support the SSX model. Each task in Rubus is defined by its: basic attributes, Task Control Block (TCB), stack/heap memory area and application code. By adding periodicity and deadline attributes to the existing task model, we are able to share all fundamental task structures between SSX tasks and the existing Blue tasks. Administration of SSX tasks is handled entirely by the Blue kernel (Figure 6.4).

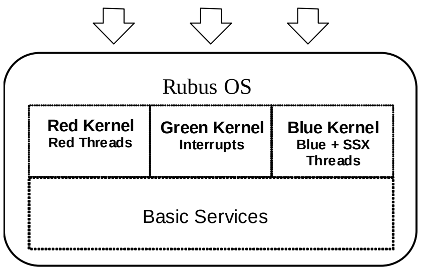

Figure 6.4: Rubus RTOS architecture with SSX model

The resulting relation of task priorities in Rubus, including SSX, is illustrated in Figure 6.5. The priority assignments and the fact that the administration of SSX tasks are handled entirely by the Blue kernel, makes the temporal attributes of tasks using the SSX model fully analyzable. In systems consisting solely of SSX tasks, the analysis can be performed with [6]. The SSX tasks can also be analyzed in hybrid systems consisting of Interrupts, Red tasks and SSX tasks with [10].

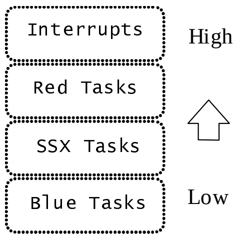

Figure 6.5: Task priorities in Rubus, including SSX

All tasks executing under the SSX model share a common stack (in fact, there is nothing that prevents stack sharing also between Red and SSX tasks). The common stack pointer, for SSX tasks, is globally accessible, hence it does not have to be stored in the TCBs. To support resource sharing in the SSX model, the immediate priority ceiling protocol was implemented. The following is a summary of all major changes made in Rubus to support the SSX execution model:

- Separation of tasks administrated by the Blue kernel and executed under different models
- Modification of administrative functions to support SSX tasks
- Error detection for SSX tasks
- Activation functionality for the SSX tasks
- Introduction of the immediate priority ceiling protocol

The integration of SSX in Rubus allows the execution model to be directly applicable for the Rubus component model. Hence, the possibility to utilize software component for applications has been extended to include four execution models, the Green, the Red, the Blue and the SSX model. The shared stack in the SSX model can be safely dimensioned, as shown below, by summing the maximum stack usage of all tasks in each priority level, and adding stack-space for interrupts. SSX tasks with equal priorities cannot pre-empt each other in Rubus, hence it suffice to take the maximum stack usage in each priority level.

$$su_{ssx} = su_{\text{int}} + \sum_{\forall i \in P} \max_{\forall \tau_j \in P_i} su_j$$

su_{ssx}, denotes maximum stack usage of all SSX tasks. su_{int}, denotes interrupt stack usage. P_i denotes the set of tasks with priority i. P denotes the set of all priority levels. τ_i denotes task i. $su(\tau_i)$ denotes stack usage, including context switch overhead, of task i. A more accurate dimensioning approach would be to examine possible pre-emptions, and identify the pre-emption(s) resulting in maximum stack usage. Identification of possible pre-emptions in a fixed priority based system is considered in [11].

6.5 Evaluation of SSX in Rubus

Stack sharing allows for an efficient memory usage, which may avoid or at least postpone the need for additional RAM in evolving systems. To illustrate how a shared stack affects memory usage, we simulate different execution scenarios where the total stack usage varies a lot depending on the execution model in use. We simulate three different execution scenarios using two different execution models, the Blue and the SSX model, for each scenario. The first scenario is obtained from a flyer promoting the SSX5 RTOS [4]. The second scenario models a traditional control application, where a sequence of tasks is used to sense, calculate control parameters, and actuate. These tasks are executed in sequence, hence they do not pre-empt each other. The third scenario illustrates a system with full pre-emption depth, i.e., all tasks are pre-empted. The scenario can be seen as an example where the benefit of SSX is less, e.g., SSX as interrupt handling tasks in systems with multiple interrupt levels.

6.5.1 Evaluation method

The evaluations are performed under a Rubus OS simulator running on a PC. We calculate the stack usage as the maximum number of data pushed onto the stack, from the dispatching of a task until it has finished execution. In doing so, we are able to include the concealed pushes of stack frames, i.e., stack usage occurring before execution of the actual task code, in the calculations. All stacks are initially filled with a pre determined data pattern. We then calculate the number of overwritten data patterns, i.e., stack usage, by examining the content of the stacks at termination of the system. It is assumed that tasks do not push frames identical to the pre determined data pattern at run time.

6.5.2 Application description

In each of the following execution scenarios, timer interrupts are generated at a frequency of 100Hz. Each timer interrupt activates the Blue kernel task, responsible for time supervision and dispatching of the tasks in the system. The service routine for timer interrupts, having highest priority in the system, execute on the stack of an active task. However, in the following scenarios, the worst-case execution times of the tasks are very short, resulting in all tasks finishing their executions before any consecutive timer interrupt hits the system. The run time model in each of the following scenarios is fixed priority, preemptive scheduling. We denote the priority of a task with Π, and its period, or in the case of sporadic tasks, its minimum interarrival time with T.

Scenario 1

The task set in the following scenario, obtained from a flyer evaluating the overheads of SSX5 [4], consists of; seven periodic tasks with periodicities ranging from 10ms to 80ms, and three interrupt handling tasks with a minimum interarrival time of 20ms (see Table 6.1).

Table 6.1: Task set, Scenario 1

Task	Π	T(ms)	Stack usage(bytes) SSX/Blue
τ_{KERNEL}	15	10	144/132
τ_1	5	10	72/152
τ_2	5	10	72/152
τ_3	4	20	72/152
τ_4	4	20	72/152
τ_5	3	40	72/152
τ_6	2	80	72/152
τ_7	2	80	72/152
τ_8	5	≥ 20	72/72
τ_9	5	≥ 20	72/72
τ_{10}	5	≥ 20	72/72

Running the system under the SSX model (with one shared stack for tasks

τ_1 - τ_{10}), results in a total stack usage of 316 bytes. With the Blue kernel stack included, the total stack usage is 460 bytes. Yet again, we evaluate scenario 1 but with the difference that tasks τ_1 - τ_7 are executed as Blue tasks (Blue execution model), achieving a pseudo periodic behavior by a call to a sleep function. According to the Rubus OS reference [7], the suspension (sleep) of the tasks requires two additional local variables, and besides the sleep call, an additional call to a function that converts the suspension time into timer ticks, resulting in increased stack usage (from 72 bytes to approximately 152 bytes) for a Blue task. This results in a total stack usage of 1480 bytes for tasks $\tau 1$ - $\tau 10$. With the Blue kernel stack included, the total stack usage is 1612 bytes. We noticed that the kernel uses 12 bytes less stack under the Blue model, than under the SSX model. This is due to Blue tasks scheduling themselves, instead of being assigned an activation time by the kernel. Table 6.2 shows the resulting stack usage for scenario 1.

Table 6.2: Stack usage, Scenario 1

Exec. model	Total stack usage (bytes)
SSX	460
Blue	1612
Savings	≈71%

Scenario 2
The following scenario consists of pure periodic tasks with harmonic period times (see Table 6.3). The scenario can be seen as a simplification of a typical vehicular control system, e.g., as described in [12].

Table 6.4 shows the resulting stack usage for scenario 2 under the SSX and Blue execution models.

Scenario 3
The previous scenario shows an ideal situation for introducing SSX tasks. However, in applications where most tasks are asynchronous and pre-emptions appear randomly, the gains of SSX tasks is less, Thus, this scenario is prepared to show that the total stack usage, in certain situations, is nearly identical between the SSX and Blue execution model. The task set in this scenario consists of one periodic task τ_4 and three event-triggered tasks τ_1 - τ_3 (see Table 6.5). The execution of the task set is prepared to exhibit full pre-emption depth meaning that if a task can be pre-empted it will be so. Each task is assigned a unique priority, thus enabling pre-emption between each pair of tasks.

Table 6.3: Task set, Scenario 2

Task	Π	T(ms)	Stack usage(bytes) SSX/Blue
τ_{KERNEL}	15	10	144/132
τ_1	5	10	72/152
τ_2	5	10	72/152
τ_3	4	20	72/152
τ_4	4	20	72/152
τ_5	3	40	72/152
τ_6	2	80	72/152
τ_7	2	80	72/152

Table 6.4: Stack usage, Scenario 2

Exec. model	Total stack usage (bytes)
SSX	216
Blue	1196
Savings	≈82%

Table 6.6 shows the resulting stack usage for scenario 3 under the SSX and Blue execution models.

6.5.3 Results

Simulations have shown that stack memory usage in Rubus OS varies when comparing systems executed under the SSX model and systems executed under the Blue model. The differences in stack usage are mainly dependent on the type of application being realized. The fact that each Blue task is allocated its own stack makes them less memory efficient in all scenarios. In an example system of 7 non-pre-emptable tasks, the difference in stack memory usage is as much as 82% less for SSX tasks than for Blue tasks. Another system derived from a flyer on SSX5, results in a difference of 71% less stack usage for the SSX tasks than for the Blue tasks. However, less difference in stack usage is observed in situations of deeply nested pre-emptions. As the pre-emption depth increases, the difference in stack usage typically decreases. This is shown by our simulations of a system with full pre-emption depth where the difference in stack usage between the SSX model and the Blue model, is relatively low.

Table 6.5: Task set, Scenario 3

Task	Π	T(ms)	Stack usage(bytes) SSX/Blue
τ_{KERNEL}	15	10	144/132
τ_1	5	-	72/72
τ_2	4	-	72/72
τ_3	3	-	72/72
τ_4	2	80	72/152

Table 6.6: Stack usage, Scenario 3

Exec. model	Total stack usage (bytes)
SSX	612
Blue	708
Savings	≈14%

Hence, the SSX model is specifically suitable for applications where jobs (or transactions) of dependent tasks are modeled without pre-emptions within the jobs e.g., control systems. On the contrary, the SSX model is less beneficial for applications experiencing large pre-emption depths. However, in any type of application, the SSX model is at least as resource efficient, with respect to stack usage, as the Blue model. This makes the SSX model an attractive choice when developing systems.

6.6 Conclusion and future work

In this paper, we presented the integration of a resource efficient and predictable single shot execution model in the Rubus RTOS. The model allows for efficient stack usage and predictability of temporal attributes. These facts make the model attractive for development of resource constrained real-time systems. The integration has shown that the model can be integrated with very simple run-time mechanisms. As future work, we are planning to include support for development and analysis (temporal and spatial) of SSX in Rubus Visual Studio (VS), which is an integrated environment for design, simulation and analyzing of embedded real-time applications.

Bibliography

[1] T.P. Baker. A stack based resource allocation policy for real-time processes. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, 1990.

[2] R. Davis, N. Merriam, and N. Tracey. How embedded applications using an rtos can stay within on-chip memory limits. In *Proceedings of the WiP and Industrial Experience Session, Euromicro Conference on Real-Time Systems*, June 2000.

[3] C.D. Locke. Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives. In *Journal of Real-Time Systems, 4*, 1992.

[4] SSX5 true RTOS Northern Real-Time Applications. Web page, http://www.ssx5.com/NRTAHome.htm.

[5] J. Xu and D.L Parnas. Priority scheduling versus pre-run-time scheduling. In *International Journal of Time-Critical Computing Systems, 18*, 2000.

[6] N. Audsley, A. Burns, R. Davis, K. Tindell, and A. Wellings. Fixed priority pre-emptive scheduling: An historical perspective. In *Real-Time Systems, 8(2/3)*, 1995.

[7] Arcticus systems. Web page, http://www.arcticus-systems.se.

[8] L. Sha, R. Rajkumar, and JP. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. In *IEEE Transactions on Computers, Volume: 39, Issue 9*, 1990.

[9] K. Sandström, C. Eriksson, and G. Fohler. Handling interrupts with static scheduling in an automotive vehicle control system. In *Proceedings of*

the 5th International Conference on Real-Time Computing Systems and Applications, 1998.

[10] J. Mäki-Turja, K. Hänninen, and M. Nolin. Efficient development of real-time systems using hybrid scheduling. In *Proceedings of the International Conference on Embedded Systems and Applications (ESA)*, June 2005.

[11] R.Dobrin and G.Fohler. Reducing the number of preemptions in fixed priority scheduling. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, July 2004.

[12] T. Riutta and K. Hänninen. Optimal design. Master's thesis, Dept. of Computer Science and Engineering, Mälardalen University, 2003.

Chapter 7

Paper D: Analysing Stack Usage in Preemptive Shared Stack Systems

Kaj Hänninen, Jukka Mäki-Turja, Markus Bohlin, Jan Carlson, Mikael Nolin
MRTC report ISSN 1404-3041 ISRN MDH-MRTC-202/2006-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, July, 2006. A version of this paper has been accepted for publication at RTSS, Rio de Janeiro, Brazil, December, 2006.

Abstract

This paper presents a novel method to determine the maximum stack memory used in preemptive, shared stack, real-time systems. We provide a general and exact problem formulation applicable for any preemptive system model based on dynamic (run-time) properties. We also show how to safely approximate the exact stack usage by using static (compile time) information about the system model and the underlying run-time system on a relevant and commercially available system model: A hybrid, statically and dynamically, scheduled system.

Comprehensive evaluations show that our technique significantly reduces the amount of stack memory needed compared to existing analysis techniques. For typical task sets a decrease in the order of 70% is typical.

7.1 Introduction

In conventional multitasking systems, each thread of execution (task) has its own allocated execution stack. In systems with a large number of tasks, a large number of stacks are required, hence the total amount of RAM needed for the stacks can grow exceedingly large.

Stack sharing is a memory model in which several tasks share one common run-time stack. It has been shown that stack sharing can result in memory savings [1, 2] compared to the conventional stack model. The shared stack model is applicable to both non-preemptive as well as preemptive systems and it is especially suitable in resource constrained embedded real-time systems with limited amount of memory. Stack sharing is currently supported by many commercial real-time kernels e.g. [3, 4, 5, 6].

The traditional method to calculate the memory requirements for a shared run-time stack in preemptive systems, is to sum the maximum stack usage of tasks in each preemption level (priority level in fixed priority systems) and possibly considering additional overheads such as memory used by interrupts and context switches. A major drawback with the traditional calculation method is that it often results in over allocation of stack memory, by presuming that all tasks with maximum stack usage in each priority level can preempt each other in a nested fashion during run-time. However, there may in many cases be no actual possibility for these tasks to preempt each other (e.g. due to explicit or implicit separation in time). Moreover, the possible preemptions may not be able to occur in a nested fashion.

Taking advantage of the fact that many real-time system exhibit a predictable temporal behavior it is possible to identify feasible preemption scenarios, i.e., which preemptions can in fact occur, and whether they can occur in a nested fashion or not. Hence, a more accurate stack analysis can be made. One example of a system that lends itself to such analysis is a hybrid, statically and dynamically, scheduled system with an off-line scheduler producing the static schedule and a fixed priority scheduler (FPS) dispatching tasks at run-time. The commercial operating system Rubus OS by Arcticus Systems AB [5], supports such a system model. The Rubus OS is mainly used in resource-constrained embedded real-time systems. For instance, in the vehicular industry, Volvo Construction Equipment (VCE) [7], BAE Systems Hägglunds [8], and Haldex Traction Systems [9] all use the Rubus OS in their vehicles and components.

In this paper we present the general problem of analyzing a shared system stack for resource constrained preemptive real-time systems. We provide a general and exact problem formulation applicable for preemptive systems based on dynamic run-time properties. We also present an approximate stack analysis method to derive a safe upper bound on stack usage in offset based (static offsets), fixed priority, preemptive systems that use a shared stack. We evaluate and show that the proposed method gives significantly lower upper bounds on stack memory requirements than existing stack dimensioning methods for fixed priority systems.

Paper outline. Section 7.2 describes related work and sets the context for the contributions of this paper. In sections 7.3, 7.4, and 7.5 we present the exact formulation of determining the maximum stack usage and our safe approximation of the stack usage for our target system model. Section 7.6 presents a simulation evaluation of our approximative analysis method, and Section 7.7 concludes the paper.

7.2 Related work

The notion of shared stack has been used in several publications to describe the ability to utilize either a common run-time stack or a pool of run-time stacks. For example, in [10] stack sharing is performed by having a pool of available stack areas. When a task starts executing, it fetches a stack from the pool, returning it at termination. In [11] Middha *et al.* address stack sharing in the sense that the stack of a task can grow into the stack area of another task.

In this paper we use the notion of stack sharing when several tasks use one common, statically allocated, run-time stack. This type of stack sharing can be efficiently implemented in systems where tasks has a run-to-completion semantics and do not self-suspend themselves. This type of stack sharing is supported by several commercial real-time operating systems e.g. [4, 5, 6]

7.2.1 Stack analysis

In [12] Baker presents the Stack Resource Protocol (SRP) that permits stack sharing among processes in static and some dynamic priority preemptive systems. The basic method to determine the maximum amount of stack usage in SRP is to identify the maximum stack usage for tasks at each priority level and then to sum up these maximums for each priority level. A safe upper bound (SPL) on the total stack usage using information about priority levels can formally be expressed as:

$$SPL = \sum_{l \in \textit{prio-levels}} \max_{i \in \textit{tasks with prio } l} (S_i) \tag{7.1}$$

where S_i is the maximum stack usage of task i.

Gai *et al.* [13] present the Stack Resource Protocol with Thresholds (SRPT) that allows stack sharing under earliest deadline scheduling. They also present an algorithm to optimize shared stack usage by use of non-preemption groups for tasks using SRPT. They extend the work of Saksena and Wang [14] by taking the stack usage of tasks into account when establishing non-preemption groups.

In [1] Davis *et al.* address stack memory requirements by using non-preemption groups to reduce the amount of memory needed for a shared stack. They show that the number of preemption levels required for typical system can be relatively small, whilst maintaining schedulability. They also state that the reduction of preemption levels are dependent on the spread of the tasks deadlines.

Although non-preemption groups can reduce the amount of RAM needed for a shared stack, the use of non-preemption groups affects a system by restricting the occurrences of preemptions, which can have a negative affect on schedulability. The method we present in this paper can further reduce the system stack by performing our analysis after preemption groups have been assigned.

7.2.2 Preemption analysis

In [15] Dobrin and Fohler presents a method to reduce the number of preemptions in fixed priority based systems. They define three fundamental conditions that have to be satisfied in order for a preemption to occur. The same conditions form the basis of our upper bound method described in Section 7.5. Note that even though the conditions are satisfied, it does not necessarily mean that a preemption will occur, only that there is a possibility for a preemption to occur. Furthermore, even though a set of preemptions are possible, it may be impossible for all of them to occur in a nested fashion, i.e., they cannot all contribute to the worst case stack usage.

Lee *et al.* [16] present a technique to bound cache-related preemption delays in fixed-priority preemptive systems. They account for task phasing and nested preemption patterns among tasks to establish an upper bound on the cache timing delay introduced by preemptions. Their work relates to ours in the sense that we investigate occurrences of nested preemption patterns. However, our objectives differ in that Lee *et al.* are mainly interested in timing delays caused by cache reloading and preemption patterns whereas we address shared memory requirements as an effect of nested preemption patterns.

7.3 Stack analysis of preemptive systems

The primary purpose of an execution stack is to store local data (variables and state registers), parameters to subroutines and return addresses. In a real-time system there is typically a separate, statically allocated stack for each task, but under certain conditions, tasks can share stack to achieve a lower overall memory footprint of the system.

We consider systems where several tasks use a common, statically allocated, run-time stack. For this to be possible, we assume that a task only uses the stack between the start time of an instance and the finishing time of that instance, i.e., no data remains on the stack from one instance of a task to the next. Furthermore, we require non-interleaving task execution, see for example [12, 1]. If υ_j starts between the start and finish of υ_i, then υ_i is not allowed to resume execution until υ_j has finished. In practice, this is ensured by not allowing tasks to suspend themselves voluntarily, or to be suspended by blocking once they have started their execution. In practice this means that OS-primitives like `sleep()` and `wait_for_event()` cannot be used, and that any blocking on shared resources must be handled before execution start, e.g., with a semaphore protocol like immediate inheritance protocol [17].

We formally define the start and finishing time of a task instance υ_i, as follows:

st_i The absolute time when υ_i actually begins executing.

ft_i The absolute time when υ_i terminates its execution.

At any given point in time, the worst case total stack usage of the system equals the sum of the stack usage for each individual task instance. Thus, with $s_i(t)$ denoting the actual stack usage of υ_i at time t, the maximum stack usage of the system can be expressed as follows:

$$\max_{t \in \textit{time instant}} \sum_{\upsilon_i \in \textit{task instances}} s_i(t). \tag{7.2}$$

This corresponds to the amount of memory that must be statically allocated for the shared stack, to ensure the absence of stack overflow errors. For some systems, e.g., non-preemptive, statically scheduled systems with simple task code, it might be possible to directly compute or estimate $s_i(t)$. In general, however, they are not directly computable before the system is executed.

We note that the total stack usage depends on three basic properties:

(i) the stack memory usage of each task instance;

(ii) the possible preemptions that may occur between any two instances; and

(iii) the ways in which preemptions can be nested.

Determining the stack memory usage of a single task instance requires knowledge of the possible control-flow paths within the task code [18]. However, due to the difficulties in determining the exact stack usage at every point in time for a given task instance, shared-stack analysis methods often assume that whenever a task is preempted, it is preempted when it uses its maximum stack depth. We make the same assumption, and use S_i to denote the maximum stack usage for task instance υ_i (thus, when υ_i and υ_j are instances of the same task, we have $S_i = S_j$). Bounds on maximum stack usage for a given task can be derived by abstract interpretation, using tools such as AbsInt [19] and Bound-T [20].

In order to calculate the maximum stack usage of the full system, we need to account for all possible preemption patterns. A task instance υ_i is preempted by another task instance υ_j if (and only if) the following holds:

$$st_i < st_j < ft_i. \tag{7.3}$$

In particular, we are interested in chains of nested preemptions. We define a *preemption chain* to be a set $\{\upsilon_1, \upsilon_2, \dots, \upsilon_k\}$ of task instances such that

$$st_1 < st_2 < \cdots < st_k < ft_k < ft_{k-1} < \cdots < ft_1. \tag{7.4}$$

Under the assumption that the worst case stack usage of a task can occur at any time during its execution, the worst case stack usage SWC for a shared stack preemptive system can be expressed as follows:

$$SWC = \max_{PC \in \textit{preemption chains}} \sum_{\upsilon_i \in PC} S_i. \tag{7.5}$$

This formulation, however, cannot be directly used for analyzing and dimensioning the shared system stack since it is based on the dynamic (only available at run-time) properties st_i and ft_i. To be able to statically analyze the system, one has to relate the static (compile-time) properties to these dynamic properties, by establishing how the system model, scheduling policy, and run-time mechanism constrain the values of the actual start and finishing times. The problem can be viewed as an optimization problem with the objective of maximizing the total stack usage of the schedule, subject to system constraints on how tasks are ordered in the schedule.

7.4 System model for hybrid scheduled systems

The system model we adopt is based on a commercial operating system Rubus OS, by Arcticus Systems AB [5], which supports the execution of both time triggered and event triggered tasks. The Rubus OS is mainly intended for and used in dependable resource-constrained embedded real-time systems.

The system model is a hybrid, static and dynamic, scheduled system where a subset of the tasks are dispatched by a static cyclic scheduler (time triggered tasks) and the rest of the tasks are dispatched by events in the system (event triggered tasks). The static schedule is constructed off-line and a fixed priority scheduler (FPS) dispatches tasks at run-time. The event triggered tasks can be categorized in two different classes: (i) interrupts which have higher priority than the time-triggered tasks, and (ii) event-triggered tasks which have lower priority than the time-triggered tasks.

The time triggered tasks share a common system stack and it is the objective of this paper to analyze, and ultimately dimension, this shared system stack efficiently. The time-triggered subsystem is used to host safety critical applications. Hence, to isolate it from any erroneous event-triggered tasks it uses its own stack.

In essence, this system model is an offset based task model with static offsets introduced by [21, 22]. In [23] we showed how tight response times can be calculated (in polynomial time [24]) for such a hybrid system.

7.4.1 Formal system model

The system model used is an offset based model with static offsets [21, 23, 24, 22] and is defined as follows: The system, Γ, consists of a set of k transactions $\Gamma_1, \ldots, \Gamma_k$. Each transaction Γ_i is activated by a periodic sequence of events with period T_i (for non-periodic events T_i denotes the minimum inter-arrival time between two consecutive events). The activating events are mutually independent, i.e., phasing between them is arbitrary.

A transaction, Γ_i, contains $|\Gamma_i|$ tasks, and each task may not be activated (released for execution) until a time, offset, elapses after the arrival of the activating event.

We use τ_{ij} to denote a task. The first subscript denotes which transaction the task belongs to, and the second subscript denotes the number of the task within the transaction. A task, τ_{ij}, is defined by a worst case execution time (C_{ij}), an offset (O_{ij}), a deadline (D_{ij}), maximum jitter (J_{ij}), maximum blocking from lower priority tasks (B_{ij}), and a priority (P_{ij}). Furthermore, S_{ij} is used to denote the maximum stack usage of τ_{ij}. The system model is formally expressed as:

$$\begin{aligned}
\Gamma :=& \{\langle \Gamma_1, T_1\rangle, \ldots, \langle \Gamma_k, T_k\rangle\} \\
\Gamma_i :=& \{\tau_{i1}, \ldots, \tau_{i|\Gamma_i|}\} \\
\tau_{ij} :=& \langle C_{ij}, O_{ij}, D_{ij}, J_{ij}, B_{ij}, P_{ij}, S_{ij}\rangle
\end{aligned}$$

There are no restrictions placed on offset, deadline or jitter, i.e., they can each be either smaller or greater than the period.

We assume that the system is schedulable and that the worst case response-time, (R_{ij}), for each task has been calculated [24]. How a scheduler can generate a feasible schedule, with interfering interrupts, is described in [25, 23].

Due to the non-interleaving criterion for stack sharing, we require that tasks exhibit a run-to-completion semantics when activated, i.e., they cannot suspend themselves. An invocation of a task can be viewed as a function call from the operating system, and the invocation terminates when the function call returns.

When tasks share the same priority they are served on a first-come first-served basis. We assume that, if access to shared resources are not handled by the static scheduler by time separation, a resource sharing protocol where blocking is done before start of execution is employed (such as the stack resource protocol [12] or the immediate inheritance protocol [17]).

The problem is to calculate the stack needed for the time triggered tasks. That is, we need to calculate the stack usage for a single transaction, which we will denote Γ_t. Task j belonging to Γ_t we will denote τ_{tj}. The tasks in the transaction can be preempted by other tasks in the transaction and by higher priority event triggered tasks.

Since Γ_t represents the static schedule, which is cyclicly repeated with period T_t, offset, jitter and deadline are less than the period, i.e., $O_{tj}, D_{tj}, J_{tj} \leq T_t$.

7.5 Stack analysis of hybrid scheduled systems

In this section we describe a polynomial time method to establish a safe upper bound on the shared stack usage for the system model described in Section 7.4. The upper bound is safe in the sense that the run-time stack can never exceed the calculated upper bound.

A safe upper-bound estimate of the exact problem can be found by using offsets and maximum response times as approximations of actual start and finishing times. Generalizing the preemption criteria identified by Dobrin and Fohler [15], we form the binary relation $\tau_{ti} \prec \tau_{tj}$ with the interpretation that τ_{ti} may be preempted by τ_{tj}. The relation holds whenever (1) τ_{ti} can become ready before τ_{tj}, (2) τ_{ti} possibly finishes (i.e., has a response time) after the start of τ_{tj}, and (3) τ_{ti} has lower priority than τ_{tj}. The relation can now formally be defined as:

$$\tau_{ti} \prec \tau_{tj} \equiv O_{ti} < O_{tj} + J_{tj} + B_{tj} \wedge O_{tj} < R_{ti} \wedge P_{ti} < P_{tj}. \qquad (7.6)$$

Lemma 1. *The $\prec$ relation is a safe approximation of the possible preemptions between tasks in Γ_t. That is, if τ_{ti} can under any run-time circumstance be preempted by τ_{tj}, then $\tau_{ti} \prec \tau_{tj}$ will hold.*

Proof of Lemma 1. *Suppose that τ_{ti} is preempted by τ_{tj}. We show that this implies (1) $O_{ti} < O_{tj} + J_{tj} + B_{tj}$, (2) $O_{tj} < R_{ti}$, and (3) $P_{ti} < P_{tj}$.*

(3) follows directly from the preemption. Now let t be the time instant when τ_{tj} has finished blocking, which implies $t \leq O_{tj} + J_{tj} + B_{tj}$. Then a possibly empty interval $[t, st_{tj}]$ of execution with higher priority than τ_{tj} follows, in which τ_{ti} cannot execute because $P_{ti} < P_{tj}$. Since τ_{ti} must start before τ_{tj}, we can conclude that $st_{ti} < t$, which together with $O_{ti} \leq st_{ti}$ and $t \leq O_{tj} + J_{tj} + B_{tj}$ gives us $O_{ti} < O_{tj} + J_{tj} + B_{tj}$ and (1). From Equation 7.3 we have $st_{tj} < ft_{ti}$ and this together with $O_{tj} \leq st_{tj}$ and $ft_{ti} \leq R_{ti}$ leads to $O_{tj} < R_{ti}$ and (2), which completes the proof. □

The upper-bound problem can now be informally stated as finding the maximum stack usage of all possible preemption chains in Γ_t. This equals finding the time instant in the schedule which has a maximum stack usage, given the approximation of actual start and finishing times with offsets and response times respectively, and assuming that at all preemptions the preempted task uses its maximal stack.

A sequence Q of tasks is a *possible preemption chain* (PPC) if it holds that $\tau_{ti} \prec \tau_{tj}$ for all τ_{ti}, τ_{tj} in Q where τ_{ti} occurs before τ_{tj} in the sequence. The stack usage SU_Q of a PPC Q is the sum of the stack usage of the individual tasks in the chain, i.e., $SU_Q = \sum_{\tau_{ti} \in Q} S_{ti}$.

A straightforward computation of a safe upper bound for a set of tasks involves computing the stack usage for all PPCs. However, for a set of n tasks there exist $2^n - 1$ different PPCs in the worst case, which yields an exponential time complexity for an algorithm based on this idea. A more efficient algorithm can be constructed by first finding sets of tasks which all overlap in time, without regarding priorities. These sets can then be investigated in turn to find a PPC with maximal stack usage.

We let the relation $\tau_{ti} \preceq \tau_{tj}$ hold whenever the semiclosed intervals $[O_{ti}, R_{ti})$ and $[O_{tj}, R_{tj})$ intersect, or more formally:

$$\tau_{ti} \preceq \tau_{tj} \equiv O_{ti} < R_{tj} \wedge O_{tj} < R_{ti}. \tag{7.7}$$

The relation $\preceq$ is a relaxation of the $\prec$ relation, that is, $\tau_{ti} \prec \tau_{tj} \rightarrow \tau_{ti} \preceq \tau_{tj}$. To see this, suppose that $\tau_{ti} \prec \tau_{tj}$ which implies $O_{ti} < O_{tj} + J_{tj} + B_{tj} \wedge O_{tj} < R_{ti}$, according to Equation 7.6. Since $O_{tj} + J_{tj} + B_{tj} \leq R_{tj}$ follows from the notion of response time, we have $O_{ti} < R_{tj} \wedge O_{tj} < R_{ti}$, which also is the definition of $\tau_{ti} \preceq \tau_{tj}$.

We can now define an *overlap set* K_r as a set of tasks where:

$$\forall \tau_{ti}, \tau_{tj} \in K_r : \tau_{ti} \preceq \tau_{tj}.$$

The stack usage SU_{K_r} of an overlap set K_r is defined as the maximum stack usage SU_Q of all PPCs Q where $Q \subseteq K_r$:

$$SU_{K_r} = \max_{\forall Q \subseteq K_r : PPC(Q)} (SU_Q). \tag{7.8}$$

K_r is *maximal* if and only if there exist no overlap set K_s such that $K_r \subset K_s$.

Lemma 2. *For any PPC Q, there exists a maximal overlap set K_r such that $Q \subseteq K_r$.*

Proof of Lemma 2. *From the definitions of a PPC and the $\prec$ and $\preceq$ relations, we know that for all tasks $\tau_{ti} \prec \tau_{tj}$ in Q it also holds that $\tau_{ti} \preceq \tau_{tj}$, and thus Q is an overlap set. Then either Q is maximal or it can become maximal by extending it with additional tasks. In either case, the lemma holds.* □

In all, the algorithm for computing the upper bound SUB on the maximum stack usage for a set of tasks Γ_t can be summarized as follows:

1. Find the maximal overlap sets in Γ_t:
   $K = \{K_1, K_2, \ldots, K_k\}$.
2. For each of them, compute SU_{K_r} as in Equation 7.8.
3. The upper bound of the stack usage for Γ_t can now be computed as follows:
$$SUB = \max_{\forall K_r \in K} (SU_{K_r}). \tag{7.9}$$

Informally, we start by finding all sets of tasks that can overlap in time based on their offsets and worst case response times, which safely approximates their actual start and finishing times. For each such set (K_i) we find all possible preemption chains (PPCs) by also taking task priorities and maximal jitter and blocking time into account, and compute the stack usage for each such chain. The stack usage of K_i is the maximum stack usage of all its PPCs, and the maximum stack usage (SUB) of the system is then obtained by taking the maximum stack usage of every K_i.

7.5.1 Correctness

In order to claim correctness of our approximate stack analysis method we have to show that it never underestimates the actual stack usage that can occur during run-time.

Theorem 1. *The value computed by the SUB algorithm is a safe upper bound on the actual worst case stack usage for tasks in Γ_t. Formally, $SWC \leq SUB$.*

Proof. Let $\Psi \subseteq \Gamma_t$ be the sequence of tasks instances participating in the preemption situation which cause the worst case stack usage, that is, $SWC = \sum_{\tau_{ti} \in \Psi} S_{ti}$. According to Lemma 1, we must have $\tau_{ti} \prec \tau_{tj}$ for tasks τ_{ti} and τ_{tj} that occur in that order in Ψ, and thus Ψ is a PPC with $SU_{\Psi} = SWC$. Then, Lemma 2 ensures that there exists a maximal overlap set K_r such that $\Psi \subseteq K_r$, and we have $SU_{\Psi} \leq SU_{K_r}$. Thus, $SWC \leq SU_{K_r} \leq SUB$, which concludes the proof. $\square$

7.5.2 Computational complexity

The relaxation of $\prec$ into interval intersection (Equation 7.7) allows us to efficiently compute an upper bound on the stack usage (Equation 7.9) by applying a polynomial longest path algorithm on the linearly-bounded number of maximal overlap sets.

To first see that the set of maximal overlap sets $K = \{K_1, K_2, \ldots, K_k\}$ contain at most n elements, i.e., $k \leq n$, consider the graph (Γ_t, E), where Γ_t is the set of vertices and $E = \{\tau_{ti}\tau_{tj} \mid (\tau_{ti} \preceq \tau_{tj}) \wedge \tau_{ti}, \tau_{tj} \in \Gamma_t\}$ is the set of edges. From Equation 7.7 we have that edges $\tau_{ti}\tau_{tj} \in E$ correspond to intersection of the semi-closed intervals $[O_{ti}, R_{ti})$ and $[O_{tj}, R_{tj})$, and therefore the graph is an *interval graph* [26]. Because every interval graph is also *chordal* [26], all maximal complete subgraphs in (Γ_t, E), which corresponds to all maximal overlap sets, can be found in linear time [27]. Furthermore, for chordal graphs there exists at most n such sets, and thus we have at most n overlap sets [26].

The problem of finding the worst PPC within a single overlap set K_r is significantly easier than for an arbitrary set of tasks. Since it holds that $\tau_{ti} \preceq \tau_{tj}$ for all tasks $\tau_{ti}, \tau_{tj} \in K_r$, and therefore in particular that $O_{ti} < R_{tj}$ for all tasks in K_r, we need only look for a maximum stack usage chain Q where (1) $O_{ti} < O_{tj} + J_{tj} + B_{tj}$, and (2) $P_{ti} < P_{tj}$ for all tasks τ_{ti} and τ_{tj} in that order in Q to find the worst PPC. A directed graph consisting of tasks in K_r and arcs corresponding to properties (1) and (2) is acyclic, and for such graphs a longest-path type algorithm can be used to find the worst PPC [28]. There exist longest-path algorithms with a time complexity of $O(n + m)$, where n is the number of tasks and m is the number of possible preemptions, of which there are at most $n(n-1)/2$. Taking the maximum of a maximal PPC in each set K_r, of which there are at most n, we will therefore find a maximum stack size PPC in at most $O(n^2 + nm)$ time.

7.6 Evaluation

We evaluate the efficiency of our proposed method to establish a safe upper bound on shared stack usage, by randomly generating realistic sized task sets. The size, load and stack usage of the task sets are derived from a wheel-loader application by Volvo Construction Equipment [7]. We use three different methods to calculate the shared system stack usage:

- SPL The traditional method to dimension a shared system stack by summing up the maximum stack usage in each priority level, see e.g. [1].
- SUB The safe upper bound on the shared stack usage presented in Section 7.5
- SLB A lower bound on on the shared stack usage, for each task set.

The lower bound is established using a simple heuristic method that tries to maximize shared stack usage by generating only feasible preemption scenarios for the task set, and thus represents scenarios that definitely can occur. From all PPCs, the heuristic selects a sample set of roughly 500 chains. For each of them, the method tries to construct a feasible arrival pattern for the ET tasks, and actual execution time values, that cause an actual preemption between the tasks in the chain. The quality of this heuristic method degrades as the length of the chains or the total number of PPCs increases, which can be seen in the figures.

By establishing a safe upper bound and a feasible lower bound, we know that the actual worst case stack usage is bounded by SUB and SLB. The reason for including SLB is to give an indication on the maximum amount of improvement there might be for SUB.

7.6.1 Simulation setup

In our simulator we generate random task sets as input to the stack analysis application. The task generator takes the following input parameters:

- Total number of TT (time triggered) tasks (default = 250)
- Total load of TT tasks (default = 60%)
- Minimum and maximum priorities of TT tasks (default = 1 and 32)
- Minimum and maximum stack usage of TT tasks (default = 128 and 2048)
- Total number of ET (event triggered) tasks (default = 8)
- Total load of ET tasks (default = 20%)
- The shortest possible minimum inter-arrival time of an ET task (default = 1.000)

The generated schedule for TT tasks is always 10.000 time units. All ET tasks have higher priority than TT tasks. The default values for the input parameters represent a base configuration derived from a real application [7].

Using these parameters a task set with the following characteristics is generated:

- Each TT offset (O_{ti}) is randomly and uniformly distributed between 0 and 10.000.

- Worst case execution times for TT tasks, C_{ti}, are initially randomly assigned between 1 and 1000 time units. The execution times gets adjusted, by multiplying all C_{ti} by a fraction, so that the the TT load (as defined by the input parameter) is obtained.
- TT priorities are assigned randomly between minimum and maximum value with a uniform distribution.

7.6.2 Results

Each diagram shows three graphs corresponding to the stack usage calculated by the three methods: SPL, SUB, and SLB. Each point in the graphs represent the mean value of 100 generated task sets. We also measured the 95% confidence interval for the mean values, these are not shown because of their small size (less than 7% of the y-value for each point). We also measured the CPU time to calculate an upper bound on shared stack usage for each generated task set. Using the method described in Section 7.5, the calculations took less than 63ms on an Intel Pentium 4, 2.8GHz with 512MB of RAM.

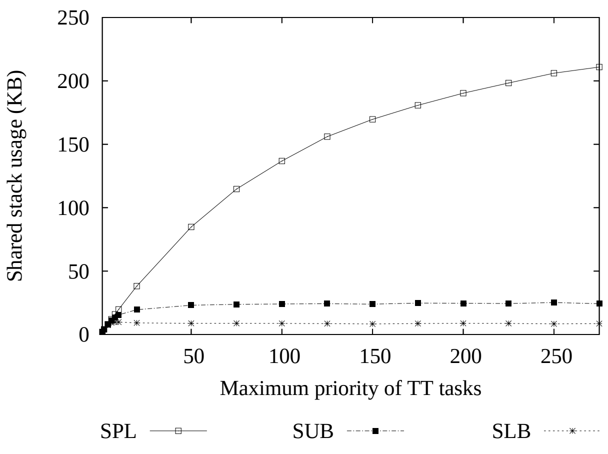

Figure 7.1: Varying the number of priority levels of TT tasks

In Fig. 7.1 we vary the maximum priority for TT tasks between 1 and 300, keeping the minimum priority at 1. This gives a distribution of possible priorities (priority levels), from 1 to n, where n is indicated by the x-axis. We see, in Fig. 7.2 which zooms in on Fig. 7.1 for maximum priorities up to 10, that the difference in stack usage between SPL and SUB is less noticeable with a low number of priority levels (see Fig. 7.2). However, for larger number of priority levels the difference is significant. SPL is not expected to flatten out before all tasks actually have unique priorities, whereas our method (SUB) flattens out significantly earlier. We conclude that the maximum number of tasks in any preemption chain is increasing very slowly (or not at all) when the number of TT tasks increases above a certain value, since the system load is constant.

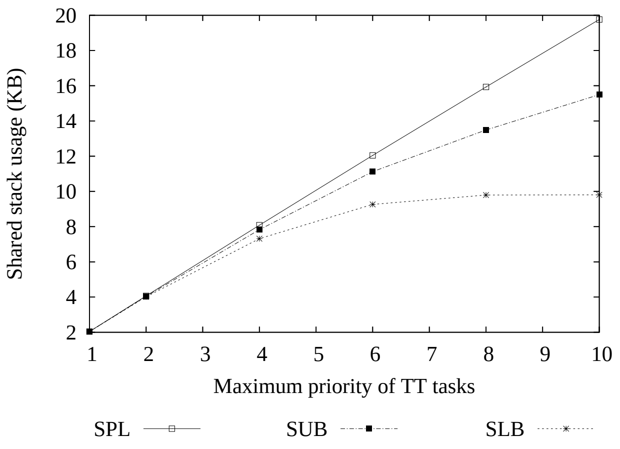

Figure 7.2: Varying the number of priority levels of TT tasks (zoom of Fig. 7.1)

In Fig. 7.3 we vary the maximum stack usage of each TT tasks between 128 bytes and 4096 bytes. We do this by assigning an initial stack of 128 bytes for each TT task, i.e. initially the stack size variation is zero. We then vary the stack size between 128 and 512 bytes, 128 and 1024 bytes, and so on. The diagram shows that SUB gives significantly lower values on shared stack usage than the traditional SPL. We also notice that an increase in stack variation scales up(linearly) the differences between SPL and SUB. The linearity is expected, since an increase in stack variation do not affect occurrences of possible preemptions in the system i.e. possible nested preemptions are retained.

In Fig. 7.4 we vary the maximum number of TT tasks between 5 and 275. We see that the shared stack usage of the traditional SPL is dramatically increasing in the beginning. This is due to the fact that when the number of TT tasks is lower than the maximum priority of TT tasks (32), most TT tasks have unique priorities. SUB, on the other hand, increases much slower than

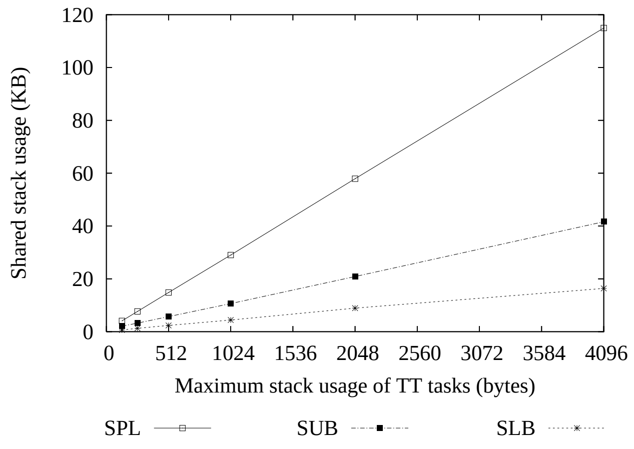

Figure 7.3: Varying stack usage of TT tasks

SPL because the maximum number of tasks involved in any preemption chain is slowly increasing. SUB is expected to further approach SPL since increasing the number of tasks will increase the likelihood of larger number of tasks involved in the preemption chains.

In Fig. 7.5 we vary the total load of TT tasks between 10% (0.1) and 70% (0.7). The figure shows that the shared stack usage of SPL is constant whereas SUB is slowly increasing. SPL is expected to be constant, since it is only affected by the number of priority levels and unaffected by the actual preemptions that can occur in a system. The increase of SUB is due to increasing response-times of TT tasks when the TT load increases, which will increase the likelihood of larger number of tasks involved in nested preemptions.

7.7 Conclusions and future work

This paper presents a novel method to determine the maximum stack memory used in preemptive, shared stack, real-time systems. We provide a general and exact problem formulation applicable for any preemptive system model based on dynamic (run-time) properties.

By approximating these run-time properties, together with information about the underlying run-time system, we present a method to safely approximate the maximum system stack usage at compile time. We do this for a relevant and

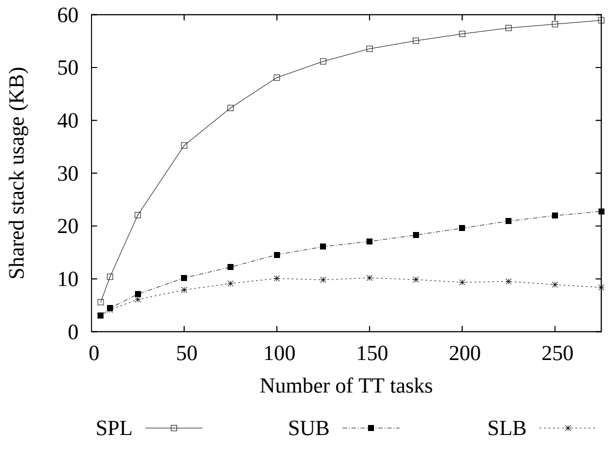

Figure 7.4: Varying the number of TT tasks

commercially available system model: A hybrid, statically and dynamically, scheduled system. Such a system model provides lot of static information that we can use to estimate the dynamic start- and finishing-times. Our approach essentially consists of finding the nested preemption pattern that results in the maximum shared stack usage. We prove that our method is a safe upper bound of the exact system stack usage and show that our method has a polynomial time complexity.

In a comprehensive simulation study we evaluated our technique, and compared it to the traditional method to estimate stack usage. We find that our method significantly reduce the amount of stack memory needed. For realistically sized task sets a decrease in the order of 70% is typical.

In this paper we focused on a system model for a given commercial real-time operating system. In the future we plan to extend our approximation method to a more general system model, to incorporate all the features of the general model for tasks with offsets [21]. Thus, making this analysis technique applicable to a wider range of systems.

Our current method could also be extended to account for other types of information that can further limit the number of possible preemptions. We currently only account for separation in time (offsets and response-times) between tasks. However, in many systems other types of information, such as precedence and mutual-exclusion relations may exists between tasks. Thus limiting

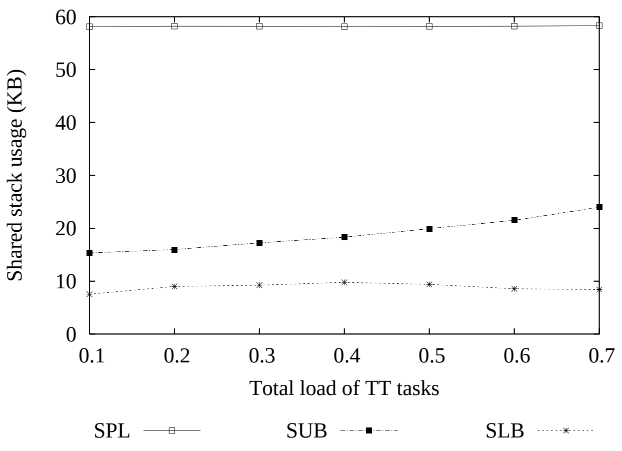

Figure 7.5: Varying the load of TT tasks

the possible preemptions.

The method presented here could also be used in synthesis and configuration tools that generate optimized systems from given application constraint. In this case, the results from our analysis can be used to guide optimization or heuristic techniques that tries to map application functionality to run-time objects.

Bibliography

[1] R. Davis, N. Merriam, and N. Tracey. How embedded applications using an rtos can stay within on-chip memory limits. In *Proceedings of the WiP and Industrial Experience Session, Euromicro Conference on Real-Time Systems*, June 2000.

[2] K. Hänninen, J. Lundbäck, K.-L. Lundbäck, J. Mäki-Turja, and M. Nolin. Efficient event-triggered tasks in an rtos. In *Proceedings of the International Conference on Embedded Systems and Applications*, June 2005.

[3] Micro Digital. Web page, http://www.smxinfo.com/mt.htm.

[4] Live Devices ETAS Group. Web page, http://en.etasgroup.com/products/rta/.

[5] Arcticus systems. Web page, http://www.arcticus-systems.se.

[6] Unicoi Systems. Web page, http://www.unicoi.com/fusion_rtos/fusion_rtos.htm.

[7] Volvo construction equipment. Web page, http://www.volvoce.com.

[8] Bae systems hägglunds. Web page, http://www.haggve.se.

[9] Haldex traction systems. Web page, http://www.haldex-traction.com/.

[10] Micro Digital Inc. *smx Features and Architecture*.

[11] B. Middha, M. Simpson, and R. Barua. MTSS: Multi task stack sharing for embedded systems. In *Proceedings of the ACM International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, San Francisco, CA, Sept 2005.

[12] T.P. Baker. A stack based resource allocation policy for real-time processes. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, 1990.

[13] P. Gai, G. Lipari, and M. Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Proceedings of the 22nd Real-Time Systems Symposium*, London, UK, Dec 2001.

[14] M. Saksena and Y. Wang. Scalable real-time system design using preemption thresholds. In *Proceedings of the 21st Real-Time System Symposium*, Dec 2000.

[15] R. Dobrin and G. Fohler. Reducing the number of preemptions in fixed priority scheduling. In *16th Euromicro Conference on Real-time Systems*, Catania, Sicily, Italy, July 2004.

[16] C. G. Lee, K. Lee, J. Hahn, Y. M. Seo, S. Lyul Min, R. Ha, S. Hong, C. Yun Park, M. Lee, and C. Sang Kim. Bounding cache-related preemption delay for real-time systems. *IEEE Transactions on Software Engineering*, 27(9):805–826, Sept 2001.

[17] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*, chapter 13.10.1 Immediate Ceiling Priority Inheritance. Addison-Wesley, second edition, 1996.

[18] R. Heckmann and C. Ferdinand. Verifying safety-critical timing and memory-usage properties of embedded software by abstract interpretation. In *Proceedings of the Design, Automation and Test in Europe*, March 2005.

[19] AbsInt. Web page, http://www.absint.com/stackanalyzer/.

[20] Tidorum. Web page, http://www.tidorum.fi/bound-t/.

[21] J. C. Palencia Gutierrez and M. Gonzalez Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proceedings of the 19th Real-Time Systems Symposium*, Dec 1998.

[22] K. Tindell. Using offset information to analyse static priority preemptively scheduled task sets. Technical Report YCS-182, Dept. of Computer Science, University of York, England, 1992.

[23] J. Mäki-Turja, K. Hänninen, and M. Nolin. Efficient development of real-time systems using hybrid scheduling. In *International conference on Embedded Systems and Applications (ESA)*, June 2005.

[24] J. Mäki-Turja and M. Nolin. Fast and tight response-times for tasks with offsets. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, July 2005.

[25] Kristian Sandström, Christer Eriksson, and Gerhard Fohler. Handling interrupts with static scheduling in an automotive vehicle control system. In *5th International Workshop on Real-Time Computing Systems and Applications (RTCSA '98)*, pages 158–165, Hiroshima, Japan, October 1998. IEEE Computer Society.

[26] T. A. McKee and F.R. McMorris. *Topics in intersection graph theory*. SIAM Monographs on Discrete Mathematics and Applications #2. Society for Industrial and Applied Mathematics (SIAM), 1999.

[27] D. J. Rose and R. E. Tarjan. Algorithmic aspects of vertex elimination. In *STOC '75: Proceedings of seventh annual ACM symposium on Theory of computing*, pages 245–254, New York, NY, USA, 1975. ACM Press.

[28] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, second edition, 2001.