SYNTHESIS AND SYNCHRONIZATION SUPPORT FOR HIERARCHICALLY SCHEDULED REAL-TIME SYSTEMS

Mikael Åsberg

2014

School of Innovation, Design and Engineering
SYNTHESIS AND SYNCHRONIZATION SUPPORT FOR HIERARCHICALLY SCHEDULED REAL-TIME SYSTEMS

Mikael Åsberg

Akademisk avhandling

som för avläggande av teknologie doktorsexamen i datavetenskap vid Akademien för innovation, design och teknik kommer att offentligen försvaras fredagen den 31 januari, 2014, 10.00 i Gamma, Mälardalens högskola, Västerås.

Fakultetsopponent: Professor Neil Audsley, University of York
Abstract

A piece of software, that we define as a software system, can consist of anything from a few lines of program code or the entire software stack in a vehicle. Software systems can be divided into smaller and partially independent parts called subsystems/partitions (we use the words partition and subsystem interchangeably). The non-functional isolation of subsystems, that appears when the software system is hierarchically divided, has great advantages when it comes to preventing fault propagation between subsystems. The hierarchical division, that we refer to as hierarchical scheduling, has other advantages as well. It facilitates re-usability and it makes timing analysis of software systems easier. Hierarchical scheduling has been shown to be a useful tool in counteracting the verification challenges that comes from the growing complexity in software. For example, the avionics specification ARINC653 and the safety-critical operating systems seL4 and PikeOS safely divide resources for independent safety-critical applications by using hierarchical scheduling.

Hierarchical scheduling can be implemented in many different ways, depending on what resource that is supposed to be shared among applications. The resource could be the CPU, memory, network etc. The work in this thesis is focused on the practical aspects of timing isolation among subsystems, i.e., sharing of the CPU resource. Hence, this work elaborates on how to adapt and extend the operating-system task-scheduler to support hierarchical scheduling. We have focused on both independent and semi-dependent subsystems. Independent subsystems only share general resources such as the CPU and memory. Semi-independent subsystems share not only the general resources, but also other logical resources that can only be accessed in a mutually exclusive way, i.e., by one subsystem at a time. An example of such a resource could be a shared memory-space, e.g., a database, a memory-mapped device etc.

This thesis has two main parts related to hierarchical scheduling: scheduler synthesis, and synchronization.

Scheduler synthesis is related to implementation and design strategies when adding support for hierarchical scheduling in an operating system. We have focused on various operating systems that were lacking the feature of hierarchical scheduling. The two most interesting operating systems that we worked on was Linux and seL4. These two operating systems represent two extremes, where Linux is more focused towards soft real-time systems and seL4 towards pure hard real-time (safety-critical) systems. Linux-based systems have in general less strict demands on correctness and more requirements on usability. Usability implies less installation efforts and less limitations in the usage of the available Linux functionality. The usability aspect is especially important for Linux systems since kernel updates occur much more frequently compared to any other operating system. Hence, extending/modifying the functionality of Linux must be done in a way that does not require any modifications to the kernel. seL4 on the other hand has strict requirements on safety, i.e., functional and non-functional correctness, but also performance efficiency. Guaranteeing correctness implies a potential loss of performance due to the added overhead that the verified software can bring. The correctness aspect includes strategies on how to verify hierarchical schedulers, but also how to minimize the scheduler overhead and achieve as good run-time performance as possible. Conclusively, there are many challenges when it comes to scheduler synthesis. There are requirements on performance, usability, correctness etc. The contribution in the synthesis part includes a scheduler framework called ExSched (External Scheduler). We have also contributed with a novel approach to verify hierarchical schedulers, and a code generator called TAtoC (Timed Automata to C) which contributes to the effective run-time performance of synthesized timed-automata models.

The second part of this thesis, synchronization, is an important general aspect of hierarchically scheduled systems since the isolation of subsystems makes resource sharing among subsystems more challenging. We have advanced the state-of-the-art in this research area by introducing a new synchronization protocol called RRP (Rollback Resource Policy) that improves on the robustness and run-time performance compared to the existing protocols. We have also conducted a large scale experimental evaluation of all existing protocols that we have implemented in the widely used real-time operating system VxWorks.

ISSN 1651-4238
Synthesis and Synchronization Support for Hierarchically Scheduled Real-Time Systems

Mikael Åsberg

2014
Time is an illusion. Lunchtime doubly so.

Douglas Noel Adams (author of *The Hitchhiker's Guide to the Galaxy*)
“Time is an illusion. Lunchtime doubly so.”

Douglas Noel Adams (author of *The Hitchhiker’s Guide to the Galaxy*)
Begreppet "realtid" förknippas ofta med att ett resultat ska presenteras snabbt, desto snabbare desto bättre. Detta kan betraktas som att något uppdateras så pass kontinuerligt och snabbt att man kan uppfatta resultatet som "fåret", alltså icke föråldrat. Ett exempel kan vara att spelresultaten från en match uppdateras i realtid på en skärm, alltså att det sker med en minimal fördröjning mellan en fysisk händelse (i detta fall ett mål) och själv uppdateringen på skärm om att denna händelse skett. Dock så finns det inge nödvändiga exakt definition på hur lång tid en "minimal fördröjning" egentligen tar.

Populärvetenskaplig sammanfattning

Begreppet "realtid" förknippas ofta med att ett resultat ska presenteras snabbt, desto snabbare desto bättre. Detta kan betraktas som att något uppdateras så pass kontinuerligt och snabbt att man kan uppfatta resultatet som "färskt", alltså icke föråldrat. Ett exempel kan vara att spelresultaten från en match uppdateras i realtid på en skärm, alltså att det sker med en minimal fördöjning mellan en fysisk händelse (i detta fall ett mål) och själva uppdateringen på skärmen om att denna händelse skett. Dock så finns det ingen riktig exakt definition på hur lång tid en "minimal fördöjning" egentligen tar.

Denna avhandling har sin grund i realtidssystem inom det vetenskapliga området Datateknik. Dessa typer av realtidssystem skiljer sig mycket från exemplet innan. De realtidssystem som behandlas i denna avhandling är ofta slutna (inbyggda system) och utför en begränsad funktion i ett större system.


Ett av dagens stora problem gällande datorsystem, inom både mjukvara och hårdvara, är att många produkter och delsystem innehåller en stor och nära ohanterlig mängd datorer och ofta är dessa sammankopplade med nätverk. Utmaningen ligger i att antalet funktioner i t.ex. en bil ökar, airbagsystemet är bara ett exempel på en funktion. Nu finns det även antisladd-system, stabiliseringsprogram, parkerings-assistans etc. vilket leder till fler och fler datorer. En bil är idag fullastad med kablage och datorer vilket ökar vikt, kostnad och även komplexiteten i dessa system.

Abstract

A piece of software, that we define as a software system, can consist of anything from a few lines of program code or the entire software stack in a vehicle. Software systems can be divided into smaller and partially independent parts called subsystems/partitions (we use the words partition and subsystem interchangeably). The non-functional isolation of subsystems, that appears when the software system is hierarchically divided, has great advantages when it comes to preventing fault propagation between subsystems. The hierarchical division, that we refer to as hierarchical scheduling, has other advantages as well. It facilitates re-usability and it makes timing analysis of software systems easier. Hierarchical scheduling has been shown to be a useful tool in counteracting the verification challenges that comes from the growing complexity in software. For example, the avionics-specification ARINC653 and the safety-critical operating systems seL4 and PikeOS safely divide resources for independent safety-critical applications by using hierarchical scheduling.

Hierarchical scheduling can be implemented in many different ways, depending on what resource that is supposed to be shared among applications. The resource could be the CPU, memory, network etc. The work in this thesis is focused on the practical aspects of timing isolation among subsystems, i.e., sharing of the CPU resource. Hence, this work elaborates on how to adapt and extend the operating-system task-scheduler to support hierarchical scheduling. We have focused on both independent and semi-dependent subsystems. Independent subsystems only share general resources such as the CPU and memory. Semi-dependent subsystems share not only the general resources, but also other logical resources that can only be accessed in a mutually exclusive way, i.e., by one subsystem at a time. An example of such a resource could be a shared memory-space, e.g., a database, a memory-mapped device etc.

This thesis has two main parts related to hierarchical scheduling: scheduler synthesis, and synchronization.
Scheduler synthesis is related to implementation and design strategies when adding support for hierarchical scheduling in an operating system. We have focused on various operating systems that were lacking the feature of hierarchical scheduling. The two most interesting operating systems that we worked on was Linux and seL4. These two operating systems represent two extremes, where Linux is more focused towards soft real-time systems and seL4 towards pure hard real-time (safety-critical) systems. Linux-based systems have in general less strict demands on correctness and more requirements on usability. Usability implies less installation efforts and less limitations in the usage of the available Linux functionality. The usability aspect is especially important for Linux systems since kernel updates occur much more frequently compared to any other operating system. Hence, extending/modifying the functionality of Linux must be done in a way that does not require any modifications to the kernel. seL4 on the other hand has strict requirements on safety, i.e., functional and non-functional correctness, but also performance efficiency. Guaranteeing correctness implies a potential loss of performance due to the added overhead that the verified software can bring. The correctness aspect includes strategies on how to verify hierarchical schedulers, but we have also focused on how to minimize the scheduler overhead and achieve as good run-time performance as possible. Conclusively, there are many challenges when it comes to scheduler synthesis. There are requirements on performance, usability, correctness etc. The contribution in the synthesis part includes a scheduler framework called \textit{ExSched} (External Scheduler). We have also contributed with a novel approach to verify hierarchical schedulers, and a code generator called \textit{TAtoC} (Timed Automata to C) which contributes to the effective run-time performance of schedulers that are synthesized from timed-automata models.

The second part of this thesis, synchronization, is an important general aspect of hierarchically scheduled systems since the isolation of subsystems makes resource sharing among subsystems more challenging. We have advanced the state-of-the-art in this research area by introducing a new synchronization protocol called \textit{RRP} (Rollback Resource Policy) that improves on the robustness and run-time performance when sharing resources, compared to the existing protocols. We have also conducted a large scale experimental evaluation of all existing protocols that we have implemented in the widely used real-time operating system VxWorks.
Acknowledgements

The two key persons that got me engaged in PhD studies and that have supported me all the way from master thesis up until now is my main supervisor Prof. Thomas Nolte and my mentor and former master-thesis supervisor Dr. Moris Behnam.

I also owe a lot of gratitude to my former study mates Alexander Casal, Amir Shariat and Arian Parsi for their encouragement during my undergraduate studies.

The PhD-student associates in my research group have given me a lot of support during my PhD studies. My greatest gratitude goes to Nima M. Khalilzad, Rafia Inam, Mohammad Ashjaei, Sara Afshar, Meng Liu, Daniel Hallmans, Hamid Faragardi and Matthias Becker.

A lot of gratitude goes to my co-authors Prof. Paul Pettersson and Dr. Shinpei Kato (Nagoya University). They have been incredibly helpful and supportive. I would also like to thank Dr. Reinder J. Bril (Eindhoven University of Technology), Clara M. Otero Pérez (NXP Semiconductors/Research), Mike Holenderski (Eindhoven University of Technology), Martijn van den Heuvel (Eindhoven University of Technology), Wim Cools (Stunf Software), Dr. Johan Kraft (Percepio AB), Dr. Insik Shin (Korea Advanced Institute of Science and Technology), Dr. Tommaso Cucinotta (Bell Laboratories/Alcatel-Lucent) and Dr. Roman Bourgade (Thales Avionics) for the interesting discussions we had and the valuable feedback that you have given me.

Many teachers at IDT gave me a lot of inspiration and encouragement during my undergraduate studies. These teachers are simply terrific! Big thanks to Åsa Lundkvist, Dag Nyström, Frank Lüders, Kristian Sandström, Mats Björkman, Christer Sandberg, Anders Pettersson, Daniel Sundmark, Johan Stärner, Mohammed El Shobaki, Jan Gustafsson, Jukka Mäki-Turja, Henrik Thane, Damir Isovic, Gordana Dodig-Crnkovic, Rikard Land, Ingrid Runnéus and Andreas Ermedahl.
I would also like to thank some of our teachers/professors at MDH for the interesting discussions and great PhD courses; Hans Hansson, Ivica Crnkovic, Sasikumar Punnekkat, Björn Lisper, Mikael Sjödin, Emma Nehrenheim, Monica Odlare, Kristina Lundqvist, Diane Pecorari and Lena Dafgård.

The administrative staff at IDT made my life as a PhD student much easier. Thanks to Carola Ryttersson, Jenny Hägglund, Susanne Fromnå and Malin Åshuvud.

I have learned a lot from the research collaboration I had with Eskilstuna ElektronikPartner AB (EEPAB). I am grateful for your support and commitment; Mikael Joki, Jimmy Hogbrink and Olof Larsson. Anders Martinsen was the main driver for this collaboration and the key person that connected me to EEPAB, thank you Anders!

Thank you so much Daniel Flemström, Lab Manager at the Industrial Research and Innovation Lab (MDH-Ericsson-ABB), for letting me use your (precious) equipment for my experiments. You have been so incredibly kind and helpful, I owe you big time!

A lot of gratitude to the wonderful people that make it so much fun to go to work! The list is long so bare with me; Abhilash, Adnan, Afshin, Aida, Alessio, Ana, Andreas G., Andreas J., Aneta, Anna, Antonio, Baran, Barbara, Batu, Carl, Christina, Eduard, Elisabeth, Federico, Fredrik, Gabriel, Gaetana, Giacomo, Guillermo, Gunnar, Hang, Holger, Hüseyin, Husni, Irfan, Jan C., Jiale, Josip, Juraj, Kan, Kivanc, Kristina F., Lars, Leo, Linus, Luka, Mahnaz, Malin R., Markus, Mehrdad, Mobyen, Nikola, Ning, Omar, Patrick, Petra, Radu, Raluca, Rikard Li., Saad, Sara Ab., Sara D., Séverine, Shahina, Stefan B., Stefan C., Svetlana, Yue, and others. My warmest gratitude also goes to the ones that left IDT but who brought a lot of joy; Aleksandar, Amine, Andreas H., Etienne, Eun-Young, Farhang, Jacqueline, Jagadish, Jiri, Jörgen, Kathrin, Rui, Shihong, Teodora and Thomas L.

This work has been supported by the Swedish Foundation for Strategic Research (Stiftelsen för strategisk forskning) and the Swedish Research Council (Vetenskapsrådet), via the HISCORE, PRESS and SYNOPSIS research programmes.

Mikael Åsberg
Västerås, January, 2014
List of publications

Papers included in the thesis


---

1The included articles are reformatted to comply with the thesis layout specifications

Other relevant publications

**Journal publications**


**Conference publications**


Workshop publications


Work-in-progress publications


Technical reports


# Contents

## I  Thesis

1  Introduction  
   1.1  Motivation and problem description  
   1.2  Goal and challenges  
      1.2.1  Goal  
      1.2.2  The design and implementation challenge (C1)  
      1.2.3  The verification and testing challenge (C2)  
      1.2.4  The synchronization challenge (C3)  
   1.3  Research methodology  
   1.4  Contributions  

2  Background  
   2.1  Real-time systems  
      2.1.1  OS task-scheduling  
      2.1.2  Resource sharing  
      2.1.3  Hierarchical real-time systems  
   2.2  Synthesis  
      2.2.1  Design and implementation  
      2.2.2  Verification, test and debug  
   2.3  Synchronization  
   2.4  Delimitations of the thesis  

3  Summary and discussion  
   3.1  Summary  
   3.2  Discussion  
      3.2.1  The design and implementation challenge  
      3.2.2  The verification and testing challenge  

xv
3.2.3 The synchronization challenge ........................................ 36
3.2.4 Satisfaction with respect to the overall goal .................... 37

4 Conclusion and future work ............................................. 41
  4.1 Conclusion .................................................................... 41
  4.2 Future work .................................................................. 42

5 Overview of the papers ......................................................... 45
  5.1 Paper A ................................................................. 45
  5.2 Paper B ................................................................. 46
  5.3 Paper C ................................................................. 47
  5.4 Paper D ................................................................. 47
  5.5 Paper E ................................................................. 48
  5.6 Paper F ................................................................. 49

Bibliography ........................................................................... 51

II Included Papers ................................................................. 57

  6.1 Introduction . ................................................................ 61
  6.2 Related work ................................................................ 62
  6.3 System model and limitations ........................................... 63
  6.4 ExSched framework ....................................................... 64
    6.4.1 User API ................................................................ 66
    6.4.2 Management of timing properties -------------------------- 66
    6.4.3 Basic approach to real-time scheduling ...................... 67
  6.5 Plug-in development ...................................................... 72
    6.5.1 Hierarchical scheduling ........................................... 72
    6.5.2 Multi-core scheduling .............................................. 74
  6.6 Experimental evaluation .................................................. 76
    6.6.1 Scheduler overhead in VxWorks ................................ 76
    6.6.2 Scheduler overhead in Linux ..................................... 78
    6.6.3 Multi-core scheduler performance ............................. 81
  6.7 Conclusion ................................................................... 86

Bibliography ........................................................................... 89
7 Paper B: 
Towards a User-Mode Approach to Partitioned Scheduling in the seL4 Microkernel 93

7.1 Introduction ...................................... 95
7.2 Preliminaries ...................................... 98
7.3 Related work ...................................... 99
7.4 Implementation ................................. 101
  7.4.1 Queue management ............................ 102
7.5 Evaluation ...................................... 104
  7.5.1 Hardware and software setup ................ 104
  7.5.2 Time measurement ............................ 105
  7.5.3 Overhead measurements ....................... 106
  7.5.4 Execution trace ............................... 109
7.6 Conclusion ...................................... 110
Bibliography ......................................... 113

8 Paper C: 
Evaluating the Run-Time Performance of Synthesised Resource-Reservation Schedulers Using TAToC, UPPAAL and Fram-C 119

8.1 Introduction ...................................... 121
8.2 Preliminaries ...................................... 123
  8.2.1 Resource reservation (CPU) ................. 123
  8.2.2 seL4 ........................................ 126
  8.2.3 Timed and task automata ..................... 127
  8.2.4 Fram-C ..................................... 128
8.3 Related work ...................................... 128
8.4 Scheduler verification and synthesis ............... 129
  8.4.1 Code synthesis ................................ 130
  8.4.2 Schedulers ................................... 134
8.5 Evaluation ...................................... 138
  8.5.1 Experimental setup ............................ 139
  8.5.2 Results ..................................... 139
8.6 Conclusion ...................................... 140
Bibliography ......................................... 142

9 Paper D: 
Modelling, Verification and Synthesis of Two-Tier Hierarchical Fixed-Priority Preemptive Scheduling 149

9.1 Introduction ...................................... 150
9.2 Preliminaries ........................................ 153
  9.2.1 Hierarchical scheduling .......................... 153
  9.2.2 Task automata and Times ........................ 154
9.3 Model .................................................. 156
  9.3.1 Global scheduler .................................. 158
  9.3.2 Event handler .................................... 158
  9.3.3 Local scheduler ................................. 159
9.4 Verification ............................................ 160
  9.4.1 Task/server systems used in the verification .... 160
  9.4.2 Global level verification ........................ 163
  9.4.3 Local level verification .......................... 167
9.5 Code synthesis ......................................... 168
9.6 Related work ........................................... 170
9.7 Conclusion ............................................. 173
Bibliography .............................................. 175

  10.1 Introduction ......................................... 183
    10.1.1 Organization of the paper ...................... 184
  10.2 Preliminaries ....................................... 185
    10.2.1 Hierarchical scheduling ........................ 185
    10.2.2 System model .................................. 185
    10.2.3 Resource sharing .............................. 186
  10.3 Related work ....................................... 190
  10.4 Rollback resource policy (RRP) .................... 192
    10.4.1 Protocol description ........................... 192
  10.5 Rollback overhead .................................. 194
    10.5.1 Task-rollback overhead ........................ 194
    10.5.2 Resource-rollback overhead ..................... 196
  10.6 Simulation results .................................. 197
    10.6.1 Simulation settings ............................ 198
    10.6.2 Schedulability ................................ 199
    10.6.3 Response time ................................ 201
    10.6.4 Resource rollback overhead ................. 204
  10.7 Conclusion .......................................... 205
Bibliography .............................................. 207
11 Paper F:
An Experimental Evaluation of Synchronization Protocol Mechanisms in the Domain of Hierarchical Fixed-Priority Scheduling 213

11.1 Introduction ...................................... 215
   11.1.1 Organization of the paper .................. 216
11.2 Preliminaries .................................... 217
   11.2.1 Hierarchical scheduling .................... 217
   11.2.2 System model ................................ 217
   11.2.3 Resource sharing in hierarchically scheduled systems 218
   11.2.4 Global/local resource sharing ............... 219
   11.2.5 Synchronization protocol mechanisms in the domain of hierarchical FPPS ................. 220
11.3 Related work .................................... 221
11.4 Implementation ................................... 223
11.5 Evaluation ....................................... 226
   11.5.1 Hardware and software setup ............... 228
   11.5.2 Principal component analysis (PCA) ........ 228
   11.5.3 Effective subsystem utilization ............ 229
   11.5.4 Deadline misses ................................ 231
   11.5.5 Protocol mechanism operations ............... 233
   11.5.6 Protocol overhead ................................ 236
11.6 Conclusion ....................................... 236
Bibliography ........................................... 239
I

Thesis
Chapter 1
Introduction
1.1 Motivation and problem description

There is an increasing competition and customer demand for more functionality in software-based products, such as cars [1], consumer electronics [2], airplanes [3, 4] etc. More functionality in the form of software makes these products more complex to develop. For example, a cell phone today has the ability to navigate, take photos, browse the internet etc., besides making phone calls. The software stack is growing rapidly since new functionality is largely implemented in software. Phone vendors cannot increase the amount of hardware at the same pace as software due to space and power restrictions. Car manufacturers experience the same kind of problem, i.e., a rapid increase in complex features and physical restrictions on how many computers that can be put in a car. The complex features come in the form of selective shock absorbers, steering assistance, electronic stability programme, braking assistance, parking assistance, collision avoidance, navigation etc. The number of on-board computers, referred to as Electronic Control Unit (ECU), as well as connecting cables must be reduced in order to conform to the strict weight and volume restrictions. Weight is for example always a limiting factor since it affects the fuel consumption. One main difference, compared to the cell phone example, is that much of the software that runs on the ECUs have real-time (and certification) requirements with strict deadlines. Take steer-by-wire as an example. This application can cause severe human casualty if it is not functioning correctly.

There are two major trends that make software integration more difficult in...
Chapter 1

Introduction

1.1 Motivation and problem description

There is an increasing competition and customer demand for more functionality in software-based products, such as cars [1], consumer electronics [2], airplanes [3, 4] etc. More functionality in the form of software makes these products more complex to develop. For example, a cell phone today has the ability to navigate, take photos, browse the internet etc., besides making phone calls. The software stack is growing rapidly since new functionality is largely implemented in software. Phone vendors can not increase the amount of hardware at the same pace as software due to space and power restrictions. Car manufacturers experience the same kind of problem, i.e., a rapid increase in complex features and physical restrictions on how many computers that can be put in a car. The complex features come in the form of selective shock absorbers, steering assistance, electronic stability programme, braking assistance, parking assistance, collision avoidance, navigation etc. The number of on-board computers, referred to as Electronic Control Unit (ECU), as well as connecting cables must be reduced in order to conform to the strict weight and volume restrictions. Weight is for example always a limiting factor since it affects the fuel consumption. One main difference, compared to the cell phone example, is that much of the software that runs on the ECUs have real-time (and certification) requirements with strict deadlines. Take steer-by-wire as an example. This application can cause severe human casualty if it is not functioning correctly.

There are two major trends that make software integration more difficult in
industries such as automotive, which has a lot of real-time requirements among its software functions. Firstly, as stated in the previous paragraph, the introduction of more features in products such as cars and cell phones increases the amount of software. The second trend is that the number of ECUs in a vehicle has reached its limits due to weight, power and space restrictions, as well as the hardware cost. The multi-core era contributes with more challenges. Before multi-core was introduced, the increasing processor frequency made it possible to integrate more software without increasing the number of processors significantly. The introduction of multi-core has stagnated the processor frequency and increased the number of cores instead. This makes it more complex to increase the amount of software due to shared memory buses and caches in the multi-core chip, even though there is plenty of processing power.

New standards such as the automotive standard AUTOSAR [1] deals with integration related challenges, i.e., the problem of integrating an increasing amount of software on a steadily decreasing number of hardware units (without violating deadlines). The avionics industry has faced the same integration related problem. It has been addressed by using the ARINC653 standard [3,4]. ARINC653 follows a strategy which divides the software into well confined containers (subsystems) in order to facilitate the integration phase. This is done by introducing time and memory partitioning. The time partitioning is essentially the same technique as hierarchical scheduling which we focus on in this thesis. The technique used in ARINC653 is also used in academic research to get predictability and composability in memory controllers [5], system-on-chip (SOC) [6] and operating systems (OSs) that are specialized for safety-critical systems [7].

We believe that the hierarchical scheduling technique has the potential to resolve problems related to integration and predictability in industry. However, there are well established industries (such as the automotive industry) that have not yet adapted to this kind of technique. The reasons for this include a potential risk that the technique itself will generate additional complexity/problems. An example of such a problem is the extra complexity to solve resource sharing when software is split apart in different subsystems. Another example is the difficulty to adapt hierarchical scheduling to fit with OSs, software standards, development processes etc. The intention with this thesis is to extend/adapt the concept of hierarchical scheduling to fit with practical needs.
1.2 Goal and challenges

This section will describe the overall goal of this thesis and three related challenges.

1.2.1 Goal

The overall goal with this thesis is to simplify the integration in complex embedded software systems with real-time requirements. In addressing the overall goal, we have identified a set of challenges related to synthesis and synchronization when using hierarchical scheduling to facilitate the integration related problems. Regarding the synthesis part, we have identified challenges related to design, implementation, testing and verification. Regarding the synchronization part, we have identified challenges inherent in resource dependencies between subsystems that are scheduled within a hierarchical scheduling framework. We will outline these challenges in more detail in the following sections.

1.2.2 The design and implementation challenge (C1)

The synthesis includes implementation challenges, however, the challenge is not just to implement hierarchical scheduling in a randomly chosen OS. There are plenty of implementations done already in academia. The real challenge is to make hierarchical scheduling practical, from an implementation perspective. The main concern is the underlying platform, i.e., the OS, on which hierarchical scheduling will be implemented. There are different design strategies regarding the locality of the scheduler that affect the practical aspects. It can be implemented in kernel space by modifying the kernel source-code [8–11], it can also be realized without kernel modifications (in kernel space) [12–14], and it can be implemented entirely in user space [15]. Observe that the referenced implementations are all based on the standard vanilla Linux OS. This is not a coincidence. A lot of ongoing research on operating systems, both in academia and industry, focus on Linux. It is an exciting field of research and it includes both hard and soft real-time approaches. Hence, we have chosen the Linux OS because it is the fastest growing OS in the domain of embedded systems [12,16]. To conclude, the aim with the synthesis, specifically targeting the implementation level, is to investigate different design options for how hierarchical scheduling can be implemented in a practical way in Linux. The meaning of "practical" is a bit vague. We refer to properties such as maintenance, installation, modularity, cost, the ability to access the entire collection of Linux
functionality etc. Maintenance is an important property and it refers to both the kernel-version compatibility and the actual functionality of the scheduler. One of the main advantages with Linux is that it provides a lot of functionality for free (if you manage to avoid license costs). Hence, license costs is of course an important factor. The available Linux functionality can be limited for tasks if they are too isolated (this applies to hard real-time only).

Another aspect to consider is the type of real-time application to aim for, i.e., soft, hard, etc.

1.2.3 The verification and testing challenge (C2)

The design and implementation part discussed previously relates to soft and hard real-time, while the verification part connects to safety-critical real-time systems. Synthesizing hierarchical scheduling for safety-critical systems typically requires some sort of verification procedure. More precisely, the questions that arise relate to the kind of verification that is needed (depends on the application domain) and how to verify hierarchical scheduling. The state-of-the-art in industry is to use ARINC653 [3] in a certified operating system such as VxWorks. The other end of the spectra is to use a 100% fully verified OS using formal methods. seL4 [7] represents such an OS. seL4 is not (yet) used in industry, in contrast with Linux which dominates the embedded-systems market. Hence, our goal of increasing the practical aspects of hierarchical scheduling does not really fit with seL4. However, we decided to embrace this OS anyway due to its unique and interesting properties. We see the potential usefulness of seL4 in the future if certification authorities decide to make the verification process more stringent.

Figure 1.1 shows how seL4 is structured. Unlike Linux which offers many ways to implement hierarchical scheduling, seL4 offers only one option: to implement it in user space. The reason for this restriction is that the kernel is verified and any modification to it requires tedious re-verification efforts. The limiting factor with a user space implementation is the potential poor performance that is inherent with it. This is something that must be addressed thoroughly in order for the scheduler to be practical.

The testing that we have conducted in this thesis is trivial once the implementation part is solved. Testing related to scheduling is basically a matter of recording schedules of tasks and subsystems and then displaying them. We have focused on the recording part since a, for our context, suitable displaying part was already available [17].
1.2 Goal and challenges

Resource sharing is one of the major limiting factors with hierarchical scheduling. This is a consequence of the isolation being the key feature and argument for hierarchical scheduling. Hierarchical scheduling has many advantages, but its Achilles heel appears when there is a need to synchronize access to resources shared between subsystems.

It is difficult to build systems without some sort of dependencies between system parts. For example, there might be devices, data-structures etc. that are shared by different subsystem parts in a system. The strict isolation of subsystems will of course make resource sharing more challenging since the isolation itself might be lost if sharing is enabled. Hence, new protocols and mechanisms must be added to hierarchical scheduling in order to preserve the isolation of subsystems in the presence of shared resources.

We started with the same reasoning in this area as we did with the design and implementation challenge (C1). The practical aspect was a key concern when we investigated this research domain. We came to the conclusion that the simplicity of the implementation of the protocol (that enabled resource sharing in a hierarchically scheduled system) was of importance. Simple protocols are generally easier to implement and use, and hence also more practical. We also focused more towards soft real-time systems (without excluding hard real-time), i.e., improving throughput. Hence, this makes the protocol suitable for both soft and hard real-time schedulers that we have developed.

![Diagram of seL4 microkernel partitioning approach](image)

Figure 1.1: The seL4 microkernel partitioning approach.

1.2.4 The synchronization challenge (C3)
1.3 Research methodology

Figure 1.2 shows the set of activities that our research work followed. We started by identifying general research problems and trends in the area of real-time systems. This activity was, to a high extent, inspired by large industries such as the avionics and automotive industry. This activity also included a study of the current state-of-the-art within the academic research community, and attending courses connected to the research field.

Identifying the research settings was the next activity to be done. This was more of an identification of technical settings, for example hardware platform (uni/multi-core) etc.

The next activity focused on sub-dividing the research problem, in combination with a particular research setting, into smaller and more manageable challenges. For example resource sharing on a uni-core platform in the presence of hierarchical scheduling. Then we continued with each sub-divided research challenge one by one. Each challenge lead to a proposed solution (after iterating with a state-of-the-art literature reading), the implementation of it, and a validation of the research results. These four steps were documented in a research article. Hence, each sub-divided research challenge eventually led to one or more research articles.

The last step, i.e., writing and publishing a research article, generally iterated back to the "Divide and redefine research problem into manageable challenges" step, giving feedback leading to modifications of the existing research challenges or the definition of new research challenges.

1.4 Contributions

The main contributions of this thesis are the following.

1: Synthesis - design and implementation. The design and implementation work in this thesis lead to a prototype scheduler called ExSched (External Scheduler). ExSched is modular, easy to install, reuse, and use. It became OS independent (supporting Linux and VxWorks) and it is freely available and open-source so that other researchers can use it. It also supports different multi-core scheduling strategies, i.e., not only hierarchical scheduling. The prototype scheduler is unique in the sense that it has all the mentioned proper-
1.3 Research methodology

Figure 1.2 shows the set of activities that our research work followed. We started by identifying general research problems and trends in the area of real-time systems. This activity was, to a high extent, inspired by large industries such as the avionics and automotive industry. This activity also included a study of the current state-of-the-art within the academic research community, and attending courses connected to the research field.

Identifying the research settings was the next activity to be done. This was more of an identification of technical settings, for example hardware platform (uni/multi-core) etc.

The next activity focused on sub-dividing the research problem, in combination with a particular research setting, into smaller and more manageable challenges. For example resource sharing on a uni-core platform in the presence of hierarchical scheduling. Then we continued with each sub-divided research challenge one by one. Each challenge lead to a proposed solution (after iterating with a state-of-the-art literature reading), the implementation of it, and a validation of the research results. These four steps were documented in a research article. Hence, each sub-divided research challenge eventually led to one or more research articles.

The last step, i.e., writing and publishing a research article, generally iterated back to the “Divide and redefine research problem into manageable challenges” step, giving feedback leading to modifications of the existing research challenges or the definition of new research challenges.

1.4 Contributions

The main contributions of this thesis are the following.

1: Synthesis - design and implementation.

The design and implementation work in this thesis lead to a prototype scheduler called ExSched (External Scheduler). ExSched is modular, easy to install, reuse, and use. It became OS independent (supporting Linux and VxWorks) and it is freely available and open-source so that other researchers can use it. It also supports different multi-core scheduling strategies, i.e., not only hierarchical scheduling. The prototype scheduler is unique in the sense that it has all the mentioned properties which also makes it a good platform to conduct empirical research on. We implemented ExSched in such a way that it satisfied our requirement on being practical. Paper A presents ExSched and it was used as a basis for the verification work (Paper B and paper C) and the evaluation part related to resource sharing (paper F).

2: Synthesis - verification and testing. The verification efforts are manifested in three papers: paper B, paper C and paper D. We answered the question regarding how to verify hierarchical scheduling in Paper D. This paper presents an implementation of a two-level hierarchical scheduler using timed automata, and a novel verification technique. The subsystems were verified independently of each other using model checking. Modularized verification is practical since modifications to a part of the system do not require re-verification of the entire system. We synthesized the scheduler for VxWorks and measured
its run-time performance. However, this preliminary scheduler showed poor run-time performance. Hence, the work in papers B and C were performed in order to improve the run-time performance. We started by adapting the implementation from paper A and integrated it to the formally verified seL4 micro-kernel (this was the work in paper B). seL4 is based on the idea of isolating components to achieve a safe system without error propagation. Introducing a hierarchical scheduler would enhance the isolation property. However, a disadvantage could be the performance loss at run-time since the scheduler must reside in user-space due to the verification of the kernel. We conducted a performance evaluation based on this (user space) scheduler implementation to estimate the run-time performance. The end goal was to integrate a verified (user-level based) hierarchical scheduler in seL4, without losing too much run-time performance. Paper C presents the performance evaluation of two different verified hierarchical schedulers in seL4. The results show a significant improvement of the run-time performance of the timed-automata based scheduler that originates from paper D. This improvement was made possible due to the introduction of a new code generator called \textit{TAToC}.

The testing part resulted in a recorder module [18] that was integrated in ExSched.

\textbf{3: Synchronization.} The last contribution is documented in papers E and F. The contribution consists of the development of a new synchronization protocol called Rollback Resource Policy (RRP). This contribution was documented in paper E. The timed-automata model from paper D was used as a basis when we evaluated RRP. The RRP protocol facilitates resource sharing within a hierarchically scheduled system. We have also evaluated the protocol thoroughly and compared it empirically against all existing protocols within the same domain (paper F). We used the scheduler that was developed in paper A, and extended the implementation with five different synchronization protocols. The results from paper F show that RRP has implementation advantages compared to the other protocols, assuming that the platform supports task rollback. We have also shown that it has good average-case performance.
Chapter 2

Background

In this chapter we present some basic concepts related to real-time systems. This includes OS scheduling, resource sharing etc. We also outline the limitations of this thesis with a list of assumed parameters.

2.1 Real-time systems

We define a real-time system as a computer system, including both hardware and software, that has strict demands on predictable timing [19]. These requirements require computer programs to finish their operations before a specific point in time. Real-time systems have of course also demands with respect to the functional correctness. These systems must provide up to 100% guarantees of meeting deadlines. This means that they may not ever exceed any of the defined deadlines.

A real-time system includes several computer programs called tasks which execute a set of operations in a sequence. The reason why the task has become the natural entity to divide a system into is mainly due to how OSs are built. These tasks can either execute in parallel together on a multi-core processor, or on a single-core processor. The latter case will result in tasks interleaving each other (called pseudo parallelism). Tasks have special attributes related to timing descriptions and these can be used in calculations [20] to check that all tasks timing demands are met before running them together. These attributes include a deadline which is the latest point in time when the task should finish its execution and a Worst Case Execution Time (WCET) which is the maxi-
mum time that a task may execute in order to complete its execution. Tasks can be triggered to run based on known points in time or due to other kind of events. Once triggered, the tasks will execute (not more than the WCET) and then wait for their next trigger (activation time). Tasks can be stopped and later resumed due to the execution of a task with priority higher than that of the interrupted task. We then say that a task preempts another task. If tasks are triggered based on known recurring points in time then we call these tasks periodic tasks [21]. The time interval between activations is defined as a period. If tasks are activated with a bounded minimum period (but possibly a larger interval) then these tasks are defined as sporadic tasks. A task that is activated at any arbitrary point in time is called an aperiodic task.

Figure 2.1 shows an example of three running tasks; T1, T2 and T3. Task T1 has the highest priority, T2 has the intermediate priority, while T3 has the lowest priority. The example shows a graphical representation of the definitions described in the previous paragraph.

There are two different kinds of real-time systems; hard real-time systems and soft real-time systems. Tasks in hard real-time systems are not allowed to miss their deadlines. However, soft real-time systems can tolerate some deadline misses. A safety-critical system is a type of real-time system which can cause catastrophic incidents if a single deadline is missed. Cars, airplanes, medical devices etc. are all examples of products that belong to the category of safety-critical systems.
2.1 Real-time systems

2.1.1 OS task-scheduling

OSs are the backbone of many real-time systems. Many OSs, for example Microsoft Windows, Linux etc., have an internal piece of software called the (task) scheduler. The job of the scheduler is to schedule tasks, i.e., decide in which order the tasks should run. If the number of tasks exceeds the available number of processors then they share the processor(s). The scheduler is responsible for scheduling the tasks according to some rule. For example, the Linux scheduler will schedule the tasks in a fair way. This means that the tasks will get an equal amount of time to run on the processor. This is accomplished by executing each task for a small amount of time, and then switch to another task for a small amount of time, and so on. This procedure continues until the last task in the list has executed its share. At this point, the scheduler starts all over again with the first task in the list. This is a scheduling rule called round robin.

Most OSs have a priority for each task. Higher priority tasks are considered more important than low priority ones. The decision of choosing the next task to run is based on the priority of the tasks. There are cases when the task priority does not ever change. We say that the scheduler schedules the tasks according to a fixed-priority scheduling rule (algorithm). The opposite case is when the scheduler dynamically alters the priorities during run-time. This type of scheduling is referred to as dynamic-priority scheduling.

A task is eligible to execute on a processor if the scheduler activates it. However, the task may only execute if it has the highest priority among all active tasks. When the activation is based on time, then the scheduler activates tasks periodically (according to a defined period). The combination of fixed-priority and periodic tasks is one of the most popular ways to schedule tasks in real-time systems.

2.1.2 Resource sharing

The processor and memory can be viewed as resources that are shared by tasks. The OS scheduler is responsible for letting tasks use the processor. The (unicycle) processor is a mutual exclusive resource in the sense that only one task can use it at a time. The access pattern to the processor is also ruled by the scheduler, i.e., it decides who is allowed to use the processor and when. The memory controller is responsible for handling the memory resource. Hence, the memory controller can also be viewed as a scheduler, but it handles the memory instead of the processor.
There are of course resources other than the processor and memory that also require mutual exclusive access. This could be for example a hardware device connected to the computer, or a piece of software etc. Most OSs have embedded support for mutual exclusive access to resources (excluding the processor and memory). This means that there is essentially a way for tasks to grab and release a lock that protects a resource. Only one task at a time can grab a lock. The release of a lock will permit another task to grab the lock. This type of lock-based mechanism is a powerful and effective way to retain mutual-exclusive access to shared resources. However, the lock-based mechanism can disrupt the priority ordering of tasks. Imagine that a low-priority task grabs a resource lock. Shortly after, a high-priority task gets activated so the scheduler decides to stop the low-priority task and let the high-priority task execute. The high-priority task decides (at some point during its execution) to grab the same lock as the one that the low-priority task is currently holding. The lock will prevent the high-priority task from continuing its execution since the lock-based mechanism prevents double access. The scheduler will let the low-priority task continue its execution instead, since the high-priority task is currently blocked by not getting access to the lock. The problem arises when a medium-priority task gets activated during the time when the low-priority task has the lock. The scheduler will of course stop the low-priority task and let the medium-priority task execute (since the medium-priority task has a higher priority than the low-priority task). This scenario represents a case when a medium-priority task executes (without holding a lock) even though a high-priority task is active. This violates the priority ordering and hence also the requirements of the scheduling algorithm. Observe that a task is allowed to execute even though other tasks with higher priority are active, if it has a lock in possession. This problem is referred to as priority inversion.

Priority-based task-scheduling requires additional support for shared resources. The lock-based mechanism is not sufficient and must be supplemented with more mechanisms in order to avoid priority inversion. The Priority Inheritance Protocol (PIP) [22] is based on the lock-based mechanism and an additional inheritance scheme that will prevent priority inversion. Relating to the example in the previous paragraph, using PIP, the low-priority task will inherit the priority of the high-priority task which will prevent the medium-priority task from running. This will solve the priority-inversion problem. However, PIP can not handle nested locks, i.e., having more than one resource locked at the same time. Assume that we have two resources called A and B. A low-priority task locks resource A, and after that, a high-priority task gets released and starts to execute. The high-priority task locks resource B but fails to lock
resource A shortly after since the low-priority task owns this lock. Hence, the high-priority task gets suspended by the scheduler until resource A is unlocked by the low-priority task. The low-priority task will inherit the priority of the high-priority task and continue its execution. At some point, the low-priority task tries to lock resource B which is currently held by the high-priority task. Hence, the low-priority task gets suspended. This situation is referred to as a deadlock since the low-priority task is waiting for resource B (locked by the high-priority task) and the high-priority task is waiting for resource A (which is held by the low-priority task).

Deadlocks can be avoided by using protocols such as the Priority Ceiling Protocol (PCP) [22] or the Stack Resource Policy (SRP) [23]. There is also a version of PCP called Immediate Priority Ceiling Protocol (IPCP) which is similar to SRP. However, we will not go into detail with IPCP since it is similar to SRP. Instead, we will focus on a more detailed description of PCP and SRP.

PCP and SRP are both ceiling-based protocols. This means that each resource in a system has a priority ceiling connected to it and this prevents deadlocks when resources are locked in a nested way. This is something that PIP lacks. The following example explains how to compute the priority ceiling of a resource. Assume that we have one resource called A and three tasks called T1, T2 and T3 that will use this resource. We can also assume that there are more tasks in the system, but which do not use resource A. Task T1 has a priority value of 3, task T2 has priority 6 and T3 has 9. The lower the priority value the higher is the priority (T1 has the highest priority). The priority-ceiling value of a resource will be the highest priority (lowest value in this case) among the tasks that will use this resource. Hence, resource A will have a priority-ceiling value of 3.

PCP and SRP calculate the priority ceiling in the same way. However, they use it in slightly different ways. The locking of a resource will raise (decrease the value of) the system ceiling. This means that the current ceiling-value of the system-ceiling will decrease, reaching the value of the ceiling of the resource that is currently being locked by a task. The initial value of the system ceiling is unset. The system ceiling value can be decreased in several steps if resources are locked one by one at different points in time which means that multiple resources are locked simultaneously. Similarly, the system-ceiling value will increase in several steps each time a resource gets unlocked. In other words, the locking of a resource can cause a decrease of the system-ceiling value, and the unlocking of a resource can cause an increase of the system-ceiling value. A task is, in the general case, not allowed to run if its priority-value is higher or equal to the system-ceiling value. The exception is of course the
task that locks a resource which causes a decrease of the system-ceiling value.
The main difference in the system-ceiling usage between PCP and SRP is that
PCP will not prevent tasks from being activated (and executed) even though
the system-ceiling value is lower than the activated task’s priority. SRP will
always prevent task preemption in cases when the system ceiling is lower than
the preempting task. SRP has in this sense a more strict ceiling rule than PCP.
SRP’s ceiling algorithm alone prevents both deadlock and priority inversion.
PCP’s ceiling algorithm only prevents deadlocks. Hence, this requires PCP to
include a second strategy, namely priority inheritance (same as PIP). This will
protect against priority inversion as well.

Figure 2.2: Example schedule using PCP.

Figure 2.2 shows an example schedule with three tasks and one resource,
together with the usage of the PCP protocol. Task T1 has the highest priority,
followed by T2, and T3 which has the lowest priority. We name the resource A,
and all tasks will use (i.e., lock and unlock) this resource in this example. Task
T3 will start executing and lock resource A. Task T1 is activated (represented
by the vertical arrow) during the time when T3 has the resource locked. T1 will
attempt to lock A but fails since T3 has it locked already. The system ceiling is
equal or lower than the priority of task T1 which prevents T1 from locking the
resource. Observe that if T1 would attempt to lock another resource that was
also shared with T3, then this lock attempt would also fail due to the system
ceiling. This will protect against deadlock due to nested resource usage. Task
T3 will inherit the priority of T1 and this prohibits priority inversion when task
T2 gets activated.

Figure 2.3 shows a schedule using SRP with the same set of tasks and resources that are used in Figure 2.2. The most obvious difference is the point in time when task T1 starts to execute. The system ceiling will stop task T1 from executing at an earlier point in time compared to PCP. SRP has a more strict approach when it comes to the implication of the system ceiling. The rule is simple and straightforward; prohibit task preemptions when the system-ceiling value is lower or equal to the task priority value. Task T2’s preemption is prohibited due to the system ceiling (and not because of the priority of T3) which prevents priority inversion. Note that SRP does not use priority inheritance like PCP. The ceiling mechanism is sufficient to prevent both priority inversion and deadlock.

Figure 2.3: Example schedule using SRP.

2.1.3 Hierarchical real-time systems

As explained previously in this chapter, a task represents the executing entity in an OS. A system is divided into a set of tasks, i.e., these tasks form the application (system) [24]. Application requirements which relate to temporal aspects (deadlines) of the functionality (called non-functional properties) are often broken down to a description of timing requirements. This could be for example a maximum delay between a physical event and a reaction from the application. Such requirements can be fulfilled by setting appropriate values to
task parameters, i.e., periods, deadlines, priorities etc. The correctness of the chosen task parameter-values can be verified by using an appropriate analysis such as the response-time analysis [20] which is specialized for fixed-priority based systems. Removing or adding tasks, as well as modifying task parameters, may affect the timing correctness. Hence, such changes may require a re-verification of the timing correctness using an analysis method.

There are real-time systems that require a more coarse-grained division of the system, i.e., where the use of tasks as the only scheduling entity becomes too fine grained. Such examples are found in, e.g., the avionics industry. There is even a standard/specification that specifies how such a coarse-grained partitioning should be done. This standard is called ARINC653 [3, 4] (Avionics Application Standard Software Interface). ARINC653 specifies a new entity called a partition (which corresponds to our description of a subsystem). A partition is a container that can contain a set of tasks. The partition protects a set of tasks from being disturbed by others in the form of memory overwrites or blocking of CPU cycles. The protection is based on well defined boundaries for processor and memory usage of partitions. Hence, this introduces two levels of abstraction. Different applications can be isolated in different partitions giving rise to protective boundaries which is safer. Each application (partition) has its own separate context, i.e., this represents the second level of the abstraction. Each partition has its own task scheduler, a set of tasks and individual timing requirements. Note that any change of a task parameter, or the number of tasks, in one partition does not affect the timing correctness of other applications residing in other partitions (since we have protective boundaries for processor and memory usage). This means that we do not have to re-analyze the timing correctness of one application if we change something in another application. ARINC653 defines specific scheduling algorithms for scheduling partitions and tasks, and how these mechanisms must be implemented in OSs in the avionics industry. ARINC653 is an example of how a coarse-grained partitioning can be done. Hierarchical scheduling is a general concept for this technique and a system is called a hierarchical system when it is divided into a hierarchical structure like this.

Figure 2.4 shows a graphical representation of this hierarchical structure that we refer to as hierarchical scheduling of real-time systems. The general case, i.e., when there is no hierarchical scheduling involved, is when the OS scheduler distributes the CPU cycles directly to tasks. The Global scheduler layer is an extra feature added to the OS (assuming that it does not already have support for hierarchical scheduling). This scheduler layer is responsible for multiplexing the CPU resource to the partitions, i.e., scheduling them accord-
18 Chapter 2. Background

Task parameters, i.e., periods, deadlines, priorities etc. The correctness of the chosen task parameter-values can be verified by using an appropriate analysis such as the response-time analysis [20] which is specialized for fixed-priority based systems. Removing or adding tasks, as well as modifying task parameters, may affect the timing correctness. Hence, such changes may require a re-verification of the timing correctness using an analysis method.

There are real-time systems that require a more coarse-grained division of the system, i.e., where the use of tasks as the only scheduling entity becomes too fine grained. Such examples are found in, e.g., the avionics industry. There is even a standard/specification that specifies how such a coarse-grained partitioning should be done. This standard is called ARINC653 [3, 4] (Avionics Application Standard Software Interface). ARINC653 specifies a new entity called a partition (which corresponds to our description of a subsystem). A partition is a container that can contain a set of tasks. The partition protects a set of tasks from being disturbed by others in the form of memory overwrites or blocking of CPU cycles. The protection is based on well defined boundaries for processor and memory usage of partitions. Hence, this introduces two levels of abstraction. Different applications can be isolated in different partitions giving rise to protective boundaries which is safer. Each application (partition) has its own separate context, i.e., this represents the second level of the abstraction. Each partition has its own task scheduler, a set of tasks and individual timing requirements. Note that any change of a task parameter, or the number of tasks, in one partition does not affect the timing correctness of other applications residing in other partitions (since we have protective boundaries for processor and memory usage). This means that we do not have to re-analyze the timing correctness of one application if we change something in another application. ARINC653 defines specific scheduling algorithms for scheduling partitions and tasks, and how these mechanisms must be implemented in OSs in the avionics industry. ARINC653 is an example of how a coarse-grained partitioning can be done.

Hierarchical scheduling is a general concept for this technique and a system is called a hierarchical system when it is divided into a hierarchical structure like this.

Figure 2.4 shows a graphical representation of this hierarchical structure that we refer to as hierarchical scheduling of real-time systems. The general case, i.e., when there is no hierarchical scheduling involved, is when the OS scheduler distributes the CPU cycles directly to tasks. The Global scheduler layer is an extra feature added to the OS (assuming that it does not already have support for hierarchical scheduling). This scheduler layer is responsible for multiplexing the CPU resource to the partitions, i.e., scheduling them according to the partition parameters. These parameters are defined in the partition Interface. The interface defines a portion/share of the CPU which is allocated to the partition. The internals (contents) of a partition can vary from a single task to a set of partitions and/or tasks. A partition can also contain a Local scheduler which is responsible for scheduling the internal tasks and/or partitions. The contents of a partition consisting of a scheduler and a set of tasks corresponds to the internals of a classical OS. The hierarchical system extends an OS with the functionality to have multiple classical real-time systems running side by side without unpredictable interference (when implemented correctly). Running multiple real-time systems together in one OS has many advantages. For example, it opens up the possibility to execute more real-time applications (in a predictable manner) with a limited number of embedded computers. This ultimately leads to less wiring and fewer computers which decreases both volume and weight. These are important factors in the automotive industry for example. In fact, the new automotive software-standard AUTOSAR [1] was developed for the purpose of simplifying the integration of a large number of software applications onto a smaller number of computers. The trend in this industry is a decrease in the number of hardware components, and an increase in the number of software functions. Hierarchical scheduling also leads to a higher degree of software reuse since software applications are packaged with
a reusable interface. The interface specifies a simple resource demand of the application which simplifies the integration with other applications. This in turn will also facilitate the use of the application in a different context.

Figure 2.5 shows an example schedule with partitions, and a set of tasks that execute inside the partitions. The partitions are assigned static priorities. Partition1 has the highest priority followed by Partition2 and Partition3. We can observe that the partitions execute in a similar pattern as tasks, i.e., periodically with preemptions. The difference compared to tasks is that partitions use a budget. The budget is a fixed duration of time which can be used by the tasks residing in the partition. Partition1 has tasks T1, T4 and T6 assigned to it. Task T9 resides in Partition2 while Partition3 includes two tasks; T2 and T7. A partition being preempted by another partition implies that the residing tasks inside the partition are also being interrupted. The task-set inside a partition is scheduled independently with respect to other tasks in other partitions. The scheduling of partitions is also independent from task scheduling within partitions. The example in Figure 2.5 represents time partitioning in ARINC653, but it is also a representation of hierarchical scheduling using two levels of scheduling.

Figure 2.5: Example schedule with three partitions.

OS vendors have adapted their OS to the avionics industry and the ARINC653 standard. One example of a popular OS is called VxWorks and it is widely used in many different industries such as automotive, avionics, aerospace etc. The VxWorks OS comes in different versions. One such version is called VxWorks 653 and it is adapted to comply with the ARINC653 standard. This
is a nice example of a technical trend, that has emerged from requirements on safety within an industry, and later was transferred to an OS that the industry uses. However, hierarchical systems are rarely found outside of the avionics industry. There are some efforts being done in the GNU/Linux OS. However, the results are limited so far (Control Groups and SCHED_DEADLINE). The automotive industry has recently adopted to the component technology in the software domain. However, there is still no development of hierarchical scheduling in the automotive software-standard AUTOSAR [1]. Academic initial efforts exist though, e.g., [25, 26].

2.2 Synthesis

This section describes the design and implementation part of our work with hierarchical scheduling, as well as software verification in general, and how it applies specifically to scheduler verification.

2.2.1 Design and implementation

Synthesis of hierarchical scheduling is essentially a matter of changing the functionality of the OS scheduler. This change can vary a bit depending on which OS that is used. It usually involves adding periodic and one-shot timer interrupts in the OS kernel. Periodic task support is not common even in real-time OSs. Hence, this functionality must be added as well. Our work is limited to periodic scheduling of partitions (and tasks), together with fixed priorities. This implies the adding of queue structures to the OS scheduler to keep track of the partition (and task) with the highest priority. Queue structures are also used to keep track of periodic interrupt events. Interrupts can be configured either one at a time or several at once. The latter implies that the OS handles these timer interrupts in an internal queue. VxWorks supports at most one pending timer interrupt at a time while Linux has support for multiple pending interrupts. To make it simple and modular, we only use one timer interrupt at a time. This makes the implementation compatible with both VxWorks and Linux (and perhaps even other OSs). Using at most one timer interrupt at a time requires an implementation of a timer queue. Our most recent developed schedulers (ExSched) use bitmaps to manage both queued timer interrupts and the priority ordering of partitions and tasks. Bitmaps offer the fastest possible operations on queues and this improves the performance of our hierarchical schedulers.
The ExSched scheduling framework offers a modular solution to scheduler development. It supports a plug-in based framework-technology where scheduler functionality can easily be reused in different schedulers. It is also easy to develop new schedulers in this framework, and test-run them. The modularity also facilitates the switching of platforms by packaging OS specific functionality in a middleware layer.

2.2.2 Verification, test and debug

Manually coded (non-verified) hierarchical schedulers fit in the soft real-time context, i.e., in areas such as video decoding in the Linux environment. Hand coded schedulers are not generally adapted to fit in the safety-critical real-time systems context. Such systems can have very high demands on safety. Safety in terms of software is, at its most extreme level, verified source-code using formal methods. seL4 is an extreme example of an OS with very high standards on safety. The entire kernel (both C-code and assembler) is formally verified using a verification technique called theorem proving. Theorem proving essentially involves both logic that describes the software to be verified, and the actual property that the user wants to verify. The logical statements are formulated by the user and then fed into a computer program which makes an attempt to solve the problem. A successful proof implies that the software has the property that was specified by the user.

The seL4 OS is a nice example of a formally proven piece of software. The certification standards of today does not require formally proven kernels. However, it would not be surprising if this would change in the future. Software is getting larger and more complex. This makes it more difficult and costly to guarantee correctness using state-of-the-practice methods. Formal methods are getting better and more efficient and they provide the safest and most reliable way to guarantee accurate properties about software.

The developers of seL4 have realized that the partitioning technique provides a way to develop safe platforms (ARINC653 has proven this). The idea is simple, divide a complex system into subsystems and provide robust boundaries between these subsystems so that they do not interfere with each other in unpredictable ways. This will seclude errors and prevent them from propagating. It will make the analysis (certification) of the entire system easier since the subsystems can be analyzed in isolation. seL4 provides the boundaries by having built-in support for (verified) memory and security partitioning. The partitioning technique and the rest of the underlying kernel has been proven to be 100% correct. The only bugs or security threats that can reside in a seL4 sys-
tem are confined in partitions which can not affect other partitions or the kernel itself. The only part that is lacking in seL4 is time partitioning, i.e., hierarchical scheduling. The reason for this is mainly due to the variability in how time partitioning can be implemented. Time partitioning can be done in many different ways using different algorithms which can vary depending on the type of application. This is the reason why the developers of seL4 have decided not to integrate hierarchical scheduling in the OS kernel. Verifying the kernel takes a lot of time. A verified kernel can not be modified without requiring a costly re-verification. In other words, changing the functionality of the scheduler is not an option since it may result in years of more verification work. The best strategy is to only keep such functionality that will not transform over time in the kernel, and put the rest in the user space (application layer). Running a scheduler in user space is possible (as the highest priority task), but it will affect all other partitions (and tasks) since it is responsible for all scheduling decisions of both partitions and tasks. Hence, it is very important that the user-space scheduler is verified for correctness. Another important aspect is performance. Schedulers running in user space will add latency to scheduling decisions, i.e., more overhead, when compared to schedulers that execute in kernel space. To conclude, a user-space scheduler that executes in a safety-critical OS such as seL4 will have requirements on its correctness and performance. The performance will depend on the efficiency of the synthesis of the scheduler. The best performance would be achieved if a manually coded scheduler would be used. Assuming it has been verified using theorem proving.

Model checking is another verification method that could be used. However, this will most likely lead to a scheduler implementation with worse performance compared to a scheduler that is verified using theorem proving. Another interesting question that arises is how to verify a hierarchical scheduler, i.e., what properties should be verified? Model-checking verification is done by exploring (searching) a model exhaustively to find properties which the user has defined using a logic-based language. The model itself is a representation of the system and the modeling language used is limited to finite-state systems. The model can optionally be translated to source code using a code generator. This translation step is critical in two aspects. The generated code must have the same behavior as the original model (this is not an uncommon problem in theorem proving as well). The source code must ideally be optimized for good run-time performance in order to be useful. The advantage with model-checking compared to theorem proving is that it is generally more user friendly. This is especially important if users are required to verify their user-space scheduler frequently due to changes in application requirements.
Conclusively, how to verify (which verification method to use) and what to verify (which properties should be verified) are two key questions when it comes to scheduler verification. Another related issue is the loss of run-time performance for which the verification process can cause (mainly due to the translation between different modeling and programming languages). Performance is an important factor for OS vendors since it will affect the entire overlying application layer. Hence, this becomes an important aspect to compete in, especially for the vendors that focus on 100% verified kernels. The reason is as follows. There is no safety standard today that requires 100% verified OSs in safety-critical applications (but this might change in the future of course). Hence, customers focus mainly on certified OSs with the best performance/cost ratio. Observe that certified OSs are not required to have every single line of code verified using formal methods. They do not even come close to the evidence of correctness compared to OSs such as seL4. On the other hand, certified OSs of today have a potential performance advantage compared to OSs like seL4. The latter group of OS vendors must hence put a lot of effort in optimizing the kernel for efficient run-time performance, if they want to be able to compete in the OS market.

When developing schedulers it is important to be able to test and debug the implementation to ensure that it performs correctly [27]. Linux has a built-in mechanism called Ftrace that can record task scheduling events in microsecond resolution with very little extra overhead. Such recordings are useful when debugging schedulers. We have included a recorder plug-in in ExSched which can record both subsystem and task scheduling events. We have also implemented processing software that can convert recorded traces from our ExSched recorder and from Ftrace, to the visualization tool Grasp [17]. Grasp is unique in the sense that it can visualize subsystems as well as tasks, but it can also display user defined events such as ready-queue manipulations, locked and unlocked resources etc. Our recorder and the processing software together with the Grasp tool has been very useful in the development of both schedulers and synchronization protocols.

### 2.3 Synchronization

The resource-sharing problem was addressed in the academic literature within real-time systems, starting with the PIP and PCP protocols in the early 1990's. There has been little advancements in this area after the introduction of the priority inheritance and ceiling protocols. Uni-core real-time systems can run ef-
Conclusively, how to verify (which verification method to use) and what to verify (which properties should be verified) are two key questions when it comes to scheduler verification. Another related issue is the loss of run-time performance for which the verification process can cause (mainly due to the translation between different modeling and programming languages). Performance is an important factor for OS vendors since it will affect the entire overlying application layer. Hence, this becomes an important aspect to compete in, especially for the vendors that focus on 100% verified kernels. The reason is as follows. There is no safety standard today that requires 100% verified OSs in safety-critical applications (but this might change in the future of course). Hence, customers focus mainly on certified OSs with the best performance/cost ratio. Observe that certified OSs are not required to have every single line of code verified using formal methods. They do not even come close to the evidence of correctness compared to OSs such as seL4. On the other hand, certified OSs of today have a potential performance advantage compared to OSs like seL4. The latter group of OS vendors must hence put a lot of effort in optimizing the kernel for efficient run-time performance, if they want to be able to compete in the OS market.

When developing schedulers it is important to be able to test and debug the implementation to ensure that it performs correctly [27]. Linux has a built-in mechanism called **Ftrace** that can record task scheduling events in microsecond resolution with very little extra overhead. Such recordings are useful when debugging schedulers. We have included a recorder plug-in in ExSched which can record both subsystem and task scheduling events. We have also implemented processing software that can convert recorded traces from our ExSched recorder and from **Ftrace**, to the visualization tool Grasp [17]. Grasp is unique in the sense that it can visualize subsystems as well as tasks, but it can also display user defined events such as ready-queue manipulations, locked and unlocked resources etc. Our recorder and the processing software together with the Grasp tool has been very useful in the development of both schedulers and synchronization protocols.

**2.3 Synchronization**

The resource-sharing problem was addressed in the academic literature within real-time systems, starting with the PIP and PCP protocols in the early 1990’s. There has been little advancements in this area after the introduction of the priority inheritance and ceiling protocols. Uni-core real-time systems can run efficiently with a deterministic worst-case behavior using these protocols. However, the ceiling-based protocol has been reused later in the literature. The starting point was the analysis of two-level hierarchical fixed-priority scheduling [28] in 2005. The same authors introduced a solution to resource sharing in two-level hierarchical fixed-priority scheduling the year after (2006). They named the new protocol HSRP [29], short for Hierarchical Stack Resource Policy. As the name suggests, they reused the SRP protocol [23] in a hierarchical environment. Most protocols that were developed after 2006 for hierarchical scheduling had the same basic concept as HSRP: the usage of SRP at each scheduling level. BROE (Bounded-delay Resource Open Environment) [30] and SIRAP (Subsystem Integration and Resource Allocation Policy) [31] are two examples of protocols (from 2007) that are also based on SRP.

The current existing synchronization protocols for hierarchically scheduled systems, e.g., HSRP, SIRAP etc., are all based on a multi-level SRP scheme as previously mentioned. This means that there are multiple SRP protocols operating (in parallel) in different scheduling levels of the system. Figure 2.6 shows the structure of a hierarchically scheduled environment with shared resources. Resources that are shared by tasks residing in different partitions are named **global resources**. **Local resources** are only shared by tasks residing
in the same partition. We can see that SRP is used in two different levels of scheduling. SRP is used to facilitate resource sharing between partitions, which means that one or several tasks in each partition among the resource-sharing partitions share a common (global) resource. We can also note that SRP is used within each partition in order to facilitate the sharing of (local) resources between tasks. The budget deplete mechanism is a protocol functionality that is important in order for resource sharing to work in the presence of partitions. This mechanism is usually the only functionality that differs between different protocols. For example, HSRP has adopted to the overrun mechanism while SIRAP implements a mechanism called skipping. We will go more into depth in this topic in the end of this Section.

Figure 2.7 illustrates an example schedule of two-level hierarchical scheduling with the SRP protocol operating both at the level of partitions and at the level of Partition3. Observe that we do not illustrate the budget deplete mechanism in this example.

We refer to the partition level when we say global level and the task level (inside partitions) refers to the local level. The example assumes that Partition3 has the lowest priority, Partition2 the middle priority and Partition1 the highest priority. Partition1 has the priority value of 3 and all partitions share one resource called A. We name resources with an upper-case letter when tasks from different partitions share it (global resource).

Observe that global resources can affect system ceilings residing both at the level of partitions and inside partitions. When global resources affect the local system-ceiling then this in turn minimizes the task interference on the task that is holding a global resource. However, the task that is holding the global resource will cause local blocking towards tasks residing in the same partition. The advantage is that this results in a reduction of the blocking effect globally at the level of partitions.

Resources that are only shared by tasks residing in the same partition are named with a lower-case letter (local resource). Resource a is a local resource that is shared by task T2 (highest priority), T3 (middle priority) and T7 (lowest priority). The priority ordering of these tasks are independent of the other tasks, i.e., they are isolated from the rest. Task T2 has the priority value of 3. Partition3 is displayed with more detail so that we can illustrate the effect of the local level SRP protocol as well.

Partition3 starts executing with task T7 running inside of it. Partition1 preempts it due to the higher priority. Task T7 locks resource A shortly after the preemption. This will cause the local SRP to raise the local ceiling inside of Partition3. In turn, this will block tasks T2 and T3. Raising the local system
ceiling can be done for two reasons. Firstly, this would be required if tasks T2 and T3 also shared resource A with task T7. Secondly, raising the local system-ceiling could also be done (even in the case if tasks T2 and T3 would never use resource A) in order to minimize the waiting time for Partition1 and Partition2. These two are prohibited to run during the time when resource A is locked, since the global SRP protocol has raised the global ceiling. The waiting time (for Partition1 and Partition2) would become longer if tasks T2 and T3 would preempt task T7 during its access to resource A.

The two SRP protocols are working independently of each other and they provide mutually exclusive access to resource A, both at the level of partitions and at the task level of Partition3. The protocols also prohibit priority inversion and deadlock at both levels. Partition3 gets preempted as soon as task T7 unlocks resource A (due to the low priority level of Partition3). The unlock will automatically activate the partitions that were blocked by resource A, hence a partition context-switch will occur. Partition1 will run first since it has the highest priority. Partition2 will run immediately after. Partition1 may
preempt **Partition2** since the latter will lock resource A after the preemption. **Partition3** will resume its execution once **Partition2** finishes due to the depletion of its budget. Task **T2** will lock a local resource called a. This resource is only shared with tasks residing in the same partition. Hence, the only ceiling that will be raised is the local one connected to **Partition3**. This ceiling raise will only affect tasks **T3** and **T7**.

The **budget deplete mechanism** solves the issue when a task holds a lock to a resource at the time when its partition’s budget depletes. Figure 2.8 illustrates this problem scenario.

![Figure 2.8: Budget depletion during the usage of a global resource.](image)

The figure shows a schedule with two partitions that share a common resource A. **Partition1** has a higher priority than **Partition2**. The task (T2) in **Partition2** does not unlock resource A before the budget depletes. We assume that the partition-level system-ceiling increases at the time of the budget depletion of **Partition2**. Hence, **Partition1** starts running immediately after the depletion. Task T1 executes for some time up until it tries to lock resource A. This resource is already locked by task T2 so **Partition1** is forced to wait (idle) until this resource is free. Another task residing in **Partition1** could potentially execute during the meantime but it does not remove the fact that T1 gets blocked for a very long time which might cause a deadline miss. **Partition1** will be scheduled to run again once task T2 unlocks the resource. The total time that will pass until this event happens could be an entire partition period-length. This is an unnecessary long time to wait for a resource. It could be prevented if a suitable mechanism would be used that could stop the budget from depleting at this improper time.

The literature proposes several different mechanisms, i.e., complete protocols (HSRP, SIRAP etc.), that can solve this issue. We will not go into depth in how these mechanisms work. However, we would like to emphasize that there is room for improvement with respect to this mechanism. Part of the research
in this thesis explores new ways that could improve the run-time performance, and make the hierarchical resource-sharing protocol more robust and safe.

A final remark regarding the assumption on the system-ceiling increase in Figure 2.8. We mentioned previously that we increase the global system-ceiling at the time when Partition2 depletes its budget. This is not compatible with the SRP protocol in the strict sense. Hence, an alternative schedule in this case (following the rules of SRP) would prevent Partition1 from running until task T2 unlocks the resource. This would lead to an even longer waiting time for task T1.

### 2.4 Delimitations of the thesis

With respect to the presented background material, the work presented in this thesis has been developed under the following assumptions:

**Real-time systems:**
Most articles included in this thesis assume hard real-time systems. The exception is paper A which presents a common platform framework that is available for the community to use. The framework is based on Linux so we relax the assumptions in this particular work to the level of soft real-time systems.

**Hardware architecture:**
We assume uni-core architectures, i.e., a single-core processor system. Our assumptions include hard real-time systems but we use hardware for our experiments which are not commonly used in hard real-time systems, e.g., Intel processors. Despite this, we assume that our developed software (schedulers etc.) should also be able to execute on hardware that have no or limited sources of unpredictable (and non-composable) behavior. Unpredictable behavior typically comes from hardware functionality in memory caches, branch-prediction components etc.

**Scheduling algorithm:**
We assume fixed-priority scheduling of periodic tasks.

**Partitioning:**
Our work includes time partitioning only.
Chapter 3

Summary and discussion

This chapter will summarize all contributions and their interconnectivity, and discuss the topics in this thesis.

3.1 Summary

Hierarchical scheduling has been supported for a long time in industries such as the avionics industry [3, 4]. However, the use of it is still restricted to this industry and has not spread much to other domains, for example to the automotive industry. The academic research within real-time systems has spent a lot of time, including both practical and theoretical work, developing the hierarchical scheduling technique. Despite this, it might just be unawareness of this technique that prevents it from being adopted in different industries. It might also be because the technique itself needs adaptations to fit with industrial needs.

The work in this thesis spans across different research areas connected to hierarchical scheduling. We believe in attacking the problem in a broader scope. This is the reason for why we have focused on challenges related to implementation, verification, testing and synchronization. The aim with this work is to deliver techniques and implementations that hopefully can be useful in order to bring hierarchical scheduling closer to industry, so that it can be considered as a usable technique suitable for, e.g., simplification of complex embedded software systems with real-time requirements.

We have also been working on many other techniques to increase the usability of hierarchical scheduling (not included in this thesis). For example,
Chapter 3

Summary and discussion

This chapter will summarize all contributions and their interconnectivity, and discuss the topics in this thesis.

3.1 Summary

Hierarchical scheduling has been supported for a long time in industries such as the avionics industry [3, 4]. However, the use of it is still restricted to this industry and has not spread much to other domains, for example to the automotive industry. The academic research within real-time systems has spent a lot of time, including both practical and theoretical work, developing the hierarchical scheduling technique. Despite this, it might just be unawareness of this technique that prevents it from being adopted in different industries. It might also be because the technique itself needs adaptations to fit with industrial needs.

The work in this thesis spans across different research areas connected to hierarchical scheduling. We believe in attacking the problem in a broader scope. This is the reason for why we have focused on challenges related to implementation, verification, testing and synchronization. The aim with this work is to deliver techniques and implementations that hopefully can be useful in order to bring hierarchical scheduling closer to industry, so that it can be considered as a usable technique suitable for, e.g., simplification of complex embedded software systems with real-time requirements.

We have also been working on many other techniques to increase the usability of hierarchical scheduling (not included in this thesis). For example,
how to adapt hierarchical scheduling to fit with the new automotive standard AUTOSAR [26], how to simplify tracing/debugging in the development of hierarchical scheduling [18], using hierarchical scheduling in an industrial case study with embedded Linux [32] etc. However, we have limited our scope and picked out three main challenges to be addressed in this thesis, with the intention to form a coherent contribution towards meeting the overall goal which is to simplify the integration in complex embedded software systems with real-time requirements.

3.2 Discussion

We will discuss the three main challenges separately with respect to our motivation and the overall goal.

3.2.1 The design and implementation challenge

Implementing hierarchical scheduling requires (in the general case) adaptations to the OS kernel. We have developed hierarchical scheduling for both Linux and seL4. We chose these two OSs because we believe in the potential usefulness of hierarchical scheduling in these two platforms. Neither of them support a practical solution for hierarchical scheduling. However, there is a potential usefulness of this type of scheduling in these platforms which makes them interesting to target.

Linux is widely used in many different technical domains which makes it the perfect host to use in order to introduce hierarchical scheduling. However, there are many implementations available from academia already. Hence, we have specialized our approach in making it simple and fast to install on different kernel versions and easy to maintain. We believe that these are key properties to have in order for hierarchical scheduling to be a practical solution to use.

Figure 3.1 shows the different approaches (related to locality) to implement schedulers in Linux, specifically related to isolating applications from each other. Approach A represents the PREEMPT_RT patch [33]. It is developed and maintained by the main Linux developers, but it requires modifications to the Linux kernel. This patch essentially makes the Linux kernel much more preemptive, and hence more suitable for hard real-time systems since latencies will become predictable (but the throughput will decline). This approach alone does not introduce isolation but it can be combined with approach B or F for example, in order to also include isolation. This will make the isolation more
solid in the sense that time partitioning gets more predictable since interrupt latencies can be bounded.

Figure 3.1: Different approaches to CPU resource partitioning.

Approach B is adopted by most implementations coming from the academic real-time systems community. The Linux kernel is open source so anyone can modify the kernel source-code and implement, e.g., hierarchical scheduling. This introduces isolation suitable for soft real-time systems since interrupts are not bounded (like in approach A).

Approach C relates to RTAI [10] and RT-Linux [11]. The HAL takes control over interrupts and relays them to specific tasks which then get hard real-time performance. These tasks are isolated from the rest of Linux in the sense that they never experience interference from other parts of Linux and they get predictable notifications from interrupts.

Approach D relates to a solution from Enea [12]. The approach is dependent on an underlying multi-core platform. Real-time tasks can be iso-
lated from non real-time tasks by putting them on a set of (shielded) cores. These cores are not affected by the interrupt interference coming from the non-shielded cores. This is done by disabling certain timers and interrupts. Hence, timing isolation is acquired through physical isolation. However, cores will interfere with each other via the memory bus and caches.

Approach E is not common but the idea is to simply put the scheduler in user space. Approach F implies putting the scheduler in a loadable kernel-module. These modules can access most primitives and data-structures inside the kernel. Kernel modules can be loaded/unloaded into the kernel at run-time on the fly.

The remaining part is to figure out which approach A-F that fits well with practical aspects. Approach A and B both suffer from serious maintenance and installation problems. Approach A requires tedious installation procedures and a lot of device-driver maintenance. Approach B requires re-modifications every time a kernel gets updated, i.e., maintenance is the main issue. Approach C can be costly since RT-Linux is not free, and RTAI limits the available Linux services that can be used by real-time tasks. Both solutions also suffer from tedious maintenance and installation procedures. Approach D is a very interesting solution since it is easily maintained and easy to install. The drawback is first of all the dependency on a hardware platform (it does not work on uni-core platforms), secondly the cost (it is a product from a company), and thirdly it limits the number of Linux primitives that are allowed to be used by tasks on shielded cores (this problem is addressed by introducing an IPC layer). Approaches E and F fit our properties best. However, observe that these techniques represent soft real-time approaches. If they were to be combined with approach A or C then they would fit hard real-time systems as well. However, there is of course a trade-off as usual when introducing a solution in Linux to hard real-time systems. Approach A for example may introduce maintenance problems related to device drivers. It also puts more burden on the user in terms of tedious installation procedures. Hence, the properties that we listed in Section 1.2.2, e.g., maintenance, installation, cost, the ability to access the entire collection of Linux functionality etc., may be satisfied for soft real-time, but hard real-time will require discarding some of these properties.

We finally chose approach F for our implementation since approach E did not support partitioning. This choice satisfied the properties that we outlined in Section 1.2.2. Our implementation also added modularity and user friendliness. The choice to create a plug-in structure definitely facilitated the framework to achieve these properties. However, we regret that we never got a chance to really test ExSched in an industrial application. Integrating ExSched
in the Android layers could have been a perfect case study.

It is not unusual that (Linux based) OS vendors only ship their user applications and kernel modules to customers, and not the kernel itself in order to avoid license payments. Hence, implementing the scheduler directly in the kernel (approach B) is not a practical approach (unless you want to spend years of work in merging your implementation in the mainline Linux kernel). Also, customers may also update their OS version often, for example via remote “swap-out” techniques. This suggests that kernel modules are more useful and flexible. The business model of shipping Linux OSs (without the actual kernel) together with a shell of application layers is not uncommon today. A good example of this is the Android OS. It runs a regular unmodified vanilla Linux kernel under the Android layer. Hierarchical scheduling has been shown to improve the energy efficiency and multimedia quality in hand-held Android devices [34]. Hence, there is a potential motivation to use ExSched in embedded systems such as Android devices. From a business perspective, it would be cheaper to let the cell-phone manufacturer load the ExSched modules into the Linux kernel they have chosen on site, instead of letting Google ship the patched kernel or force the manufacturer to patch the kernel themselves.

We realize that many pieces are missing in order for this framework to become complete, for example resource sharing. Making our ExSched project free and open source will hopefully attract users from academia so that they can contribute with more functionality to ExSched.

### 3.2.2 The verification and testing challenge

We also prototyped a hierarchical scheduler for the seL4 OS. seL4 is an academic prototype OS designed with extreme requirements on safety and security. Its industrial use is limited today but it will most likely outperform existing OSs in the safety aspect in the future. It is designed with partitioning as a basis for safety but it currently lacks a proper time partitioning mechanism. It is up to the user to implement hierarchical scheduling if needed. The way that seL4 is structured requires a performance efficient scheduler. This is something that we had in mind when we developed a hierarchical scheduler for seL4. We experimented with different scheduler implementations and different verification techniques. Our conclusion is that it is far more tedious to use theorem proving compared to model checking. However, we discovered at first that it was straightforward to verify hierarchical scheduling when using timed automata and model checking, but the run-time performance suffered a lot. Good performance is something you get for free when you use theorem proving, but we
still wanted to proceed with trying to improve the run-time performance of the generated C-code from timed automata. The final results showed that we could improve the run-time performance slightly by optimizing the code generator in UPPAAL and by simplifying the timed-automaton model of the scheduler. The biggest disappointment was to observe a tiny performance difference between TAtoC (our new code generator) and the default code generator in the UPPAAL-TIMES tool when running experiments in seL4.

The use of theorem proving was a dead end. We used the Frama-C framework to verify the C-code of a hierarchical scheduler. The verification process became far too complex and tedious, but we succeeded in the end. Hence, we still believe in using timed automata and model checking, but there is still work to be done. It would be interesting to perform an evaluation of TAtoC in a different platform, to see if the run-time performance of the generated code from TAtoC would remain the same, improve or deteriorate.

Adding the scheduler recorder plug-in in ExSched facilitated the testing of schedulers. This recorder was helpful for debugging the schedulers and synchronization protocols that we developed.

3.2.3 The synchronization challenge

Our hypothesis is that hierarchical scheduling can facilitate integration challenges in complex embedded real-time systems. However, it needs adaptations to fit with general system requirements. An example of such is the migration from uni-core to multi-core systems. For example, shared bus and cache-memories will disrupt the isolation of time partitioning. There are of course other aspects to consider as well. We have instead focused on the synchronization challenge in hierarchically scheduled systems on uni-core platforms. The problem itself is avoided in, for example, the avionics industry by not allowing synchronization between partitions. The reason is that such synchronization would add complexity in both the implementation and the certification process. Another issue is that there are very few studies on this topic. This is why we have chosen to make a contribution to this research area. Specifically, we have focused on developing a new synchronization technique (the Rollback Resource Policy) in order to decrease the average response-time of tasks, the implementation complexity, and improve on the robustness of the protocol itself. We focused mostly on the implementation aspects of the protocol and the average-case performance (rather than the worst-case performance). The reason for this is mainly due to that RRP has exactly the same worst-case analysis as the SIRAP protocol, i.e., there was not much more contribution that could
be added to the analysis part. Instead, we looked at the implementation aspects of both RRP and the other protocols (HSRP, SIRAP and HSTP). The most surprising result was the low overhead of RRP compared to the other protocols, despite the fact that RRP uses a fairly expensive OS mechanism, i.e., the task rollback. We should also note that these results are limited to the VxWorks OS. We also observed good throughput. Comparing to the other protocols, RRP could schedule most task configurations that we tested without any deadline violations. On the other hand, RRP was designed to be optimistic so this was perhaps expected. The only drawback with RRP is that the use of it requires that resources must be possible to abort, and rolled back to a consistent state if necessary. However, we do not know to what extent this is a problem, since the effects are application dependent.

To conclude, we found more implementation disadvantages with HSRP and SIRAP, comparing to RRP and HSTP. HSRP suffers from the overrun mechanism since overruns cannot be smaller than a scheduling tick. Any task that unlocks a resource in less than one OS scheduling tick could potentially lead to an unused portion of the overrun time which creates unnecessary interference on other subsystems and tasks. The severity of this problem is dependent on the difference in length between critical sections and the OS scheduling tick. SIRAP suffered even more since its mechanism, the skipping operation, requires a safe margin of time to be added to the worst-case estimated critical-section length. The extended length of the critical section had a massive negative impact on the run-time performance of SIRAP. These implementation aspects are not reflected in the analysis of the protocols. They might even require modifications to the existing schedulability analysis of the protocols. Hence, by the experimental study we gained a lot of new insights regarding this matter.

### 3.2.4 Satisfaction with respect to the overall goal

In Section 1.2.1 we identified the overall goal of this thesis as “to facilitate integration in complex embedded systems with real-time requirements”. We focused on the uni-core platform, which restricted the scope towards a technical solution of using hierarchical scheduling of the CPU. The multi-core platform inherits the same challenges as with the uni-core hardware, i.e., assuming that the number of applications are more than the number of cores, then integration will take place on each core in the same manner as with the uni-core case. However, multi-core platforms have additional challenges related to shared memory buses, caches etc. Many-core systems will most probably not have exactly the same integration challenges as with uni-core platforms,
but they will face the same challenges that are present with current multi-core platforms. In other words, we believe that solutions to integration related challenges on uni-core platforms are also applicable on multi-core platforms, to a certain degree. Hence, we believe that the choice to use hierarchical scheduling satisfies the overall goal and that it still targets a general set of hardware platforms (excluding many-core platforms). The shared bus/cache challenge is another more general problem not related to the integration challenge only.

We focused on the hierarchical scheduling technique as a basis for tackling the problem with integration in complex embedded real-time systems. We then identified challenges related to the use of hierarchical scheduling in such a context. The challenges identified were related to synthesis and synchronization aspects of hierarchical scheduling. The synthesis part included design and implementation aspects such as the choice of an OS, maintenance, scheduler locality, cost, correctness/verification etc.

ExSched was the solution we came up with that relates to the design and implementation challenge. It relates to embedded systems in the sense that ExSched is mainly focused on the Linux OS which is one of the major platforms for embedded systems. It is intended to be a practical solution, however, the judge of this will be the number of users in the future. ExSched also served its secondary purpose as being a base prototype scheduler-platform to facilitate further research on hierarchical scheduling.

ExSched supported the verification part of our synthesis direction. Embedded systems with real-time requirements typically include safety-critical systems as well. We connected hierarchical scheduling to this line of research by looking at verification techniques of the scheduling algorithm itself. We further extended the work by focusing on the run-time performance of verified hierarchical schedulers. Run-time performance is always an important aspect in embedded systems since these systems usually have limited resources such as CPU, memory etc.

The synchronization challenge is embedded in the hypothesis to use hierarchical scheduling. Another equivalent technique, although limited to the multi-core platform, is called core shielding (see Figure 3.1 in Section 3.2.1). The synchronization challenge in this setting is different [35] from the one we focused on. Our solution to the synchronization challenge is a protocol called the Rollback Resource Policy (RRP). The marketing strategy of RRP with respect to embedded systems in general, is its good average-case performance, i.e., the throughput. It can also potentially improve the worst-case behavior since it is inherent with favorable implementation properties. The protocol is based on an optimistic approach (rollback) which has a tendency to utilize the
3.2 Discussion

CPU better than many of the existing protocols. Hence, the effective run-time performance of the protocol is well suited for resource-limited embedded systems.
Chapter 4

Conclusion and future work

This chapter will conclude our contributions and discuss future-work directions.

4.1 Conclusion

The research results in this thesis can be summarized in the following way. The main contributions of this thesis is the scheduling framework ExSched (paper A), efficient synthesis of verified hierarchical schedulers including the development of a code generator called TAtoC for timed automata (paper B and C), a novel verification technique for fixed-priority preemptive hierarchical scheduling (paper D), and the synchronization protocol RRP including evaluations (paper E and F).

We have developed a free (open-source) scheduler framework for Linux called ExSched. It has good run-time performance (compared to other academic schedulers) and it has the potential to be adopted by industry due to its simplified installation process. A version of this scheduler has been developed especially for the seL4 OS. All of our developed schedulers are specialized in some way, either by being modular and installation efficient, or proven correct by using our novel verification technique. These factors may be perceived as performance degrading. However, we have collected performance results from all of our developed schedulers and showed that the performance is comparable to equivalent schedulers. The improved performance was accomplished by, for example, using advanced queue structures and our effective code generator.
Chapter 4

Conclusion and future work

This chapter will conclude our contributions and discuss future-work directions.

4.1 Conclusion

The research results in this thesis can be summarized in the following way. The main contributions of this thesis is the scheduling framework ExSched (paper A), efficient synthesis of verified hierarchical schedulers including the development of a code generator called TAtoC for timed automata (paper B and C), a novel verification technique for fixed-priority preemptive hierarchical scheduling (paper D), and the synchronization protocol RRP including evaluations (paper E and F).

We have developed a free (open-source) scheduler framework for Linux called ExSched. It has good run-time performance (compared to other academic schedulers) and it has the potential to be adopted by industry due to its simplified installation process. A version of this scheduler has been developed especially for the seL4 OS. All of our developed schedulers are specialized in some way, either by being modular and installation efficient, or proven correct by using our novel verification technique. These factors may be perceived as performance degrading. However, we have collected performance results from all of our developed schedulers and showed that the performance is comparable to equivalent schedulers. The improved performance was accomplished by, for example, using advanced queue structures and our effective code generator
We have focused the synchronization work towards the development of a new protocol called RRP (Rollback Resource Policy). We have performed extensive evaluations of this protocol by comparing it to all other equivalent protocols within the area of fixed-priority hierarchical scheduling. Our simulations have shown that RRP is (CPU) resource efficient. In fact, it generated the least amount of blocking compared to the other protocols when we experimented on more than a quarter of a million task configurations [36]. The implementation study has shown that RRP has low overhead (the second best) and several implementation advantages compared to the other protocols. The only disadvantage with RRP is that it is only compatible with shared resources that can be aborted and rolled back to a consistent state.

These contributions have responded to the challenges that we specified when we chose to reach the overall goal by using hierarchical scheduling. In summary, the challenges addressed in this thesis constitute together an important step towards fulfilling our overall goal: to simplify the integration in complex embedded systems with real-time requirements.

4.2 Future work

We see a great potential with ExSched and the future development of it. There has been interest of it in both academia [37, 38] and industry [32]. Possible extensions could be to include resource-sharing protocols and extend the number of supported platforms. However, we see most potential of its usage in the Linux environment. An industrial case-study (using Linux) would be interesting to perform, for example on the Android platform.

The verified schedulers that we have developed are complex to implement and difficult to reuse due to the verification process. We predict that verified OSs (such as seL4) will leave it to the user to handle the scheduling of tasks (and hence also the verification part), since scheduling is dependent on the application. A similar approach as ExSched could be useful in this context, i.e., a modular scheduler framework with plug-in support which makes it easy to reuse scheduler functionality in new schedulers. The added dimension is the verification process, i.e., how to sort out which properties that are suitable to verify and how to reuse them in a safe and correct way without adding too much extra verification efforts.

Using hierarchical scheduling, in a verified context or not, will most likely require some sort of resource sharing between partitions. We have looked at the
run-time efficiency and schedulability of existing resource-sharing protocols and even developed a new one (RRP). All protocols have been implemented and evaluated. The remaining part is to perform an actual real-life study, i.e., look at the practicality of the protocols when operating on actual shared resources in an industrial context. The most practical way towards this study would be to integrate these protocols in ExSched.

We also see a potential continuation of RRP by mitigating its limitation on the type of resources that it is compatible with. Assuming that all resources are not compatible with abort and rollback, a potential solution could be to include an alternative mechanism to use with these types of resources. The resulting protocol would hence be a fusion of different protocols with new properties and a new schedulability analysis. A possible extension to the protocol could be to use a schedule predictor instead of the rollback mechanism. It could use task instrumentation and queue information to decide whether it is safe to lock a resource or not. This would completely solve the resource-type limitation of RRP.
Chapter 5
Overview of the papers

5.1 Paper A
Mikael ˚Asberg, Shinpei Kato, Thomas Nolte and Ragunathan Rajkumar,
ExSched: An External CPU Scheduler Framework for Real-Time Systems,
In proceedings of the 18th IEEE International Conference on Embedded and
Real-Time Computing Systems and Applications (RTCSA), pages 240-249,
August, 2012.

Summary
Scheduling theory and algorithms have been well studied in the
real-time systems literature. Many useful approaches and solutions have ap-
peared in different problem domains. While their theoretical effectiveness has
been extensively discussed, the community is now facing implementation chal-
lenges that show the impact of the algorithms in practice.

In this paper, we propose a scheduler framework, called
ExSched, which enables different schedulers to be developed for different OS platforms with-
out any modifications to the OS itself, using a unified interfa-
ce. The framework will easily keep up with changes in the kernel since it is only dependent on a
few kernel primitives. The usefulness of this framework is that scheduling
policies can be implemented as external plug-ins. They can sim-
ply use the ExSched interface instead of platform-dependent functions, since platform de-
tails are abstracted by ExSched. The advantage for industry is that they would
more easily keep up with new kernel versions since ExSched does not require
patches. The advantage for academia is that we could focus on the devel-
opment of schedulers instead of tedious and time-consuming installations of
patched kernels.

Our prototype implementation of ExSched supports Linux and VxWorks
Chapter 5

Overview of the papers

5.1 Paper A


Summary  Scheduling theory and algorithms have been well studied in the real-time systems literature. Many useful approaches and solutions have appeared in different problem domains. While their theoretical effectiveness has been extensively discussed, the community is now facing implementation challenges that show the impact of the algorithms in practice.

In this paper, we propose a scheduler framework, called ExSched, which enables different schedulers to be developed for different OS platforms without any modifications to the OS itself, using a unified interface. The framework will easily keep up with changes in the kernel since it is only dependent on a few kernel primitives. The usefulness of this framework is that scheduling policies can be implemented as external plug-ins. They can simply use the ExSched interface instead of platform-dependent functions, since platform details are abstracted by ExSched. The advantage for industry is that they would more easily keep up with new kernel versions since ExSched does not require patches. The advantage for academia is that we could focus on the development of schedulers instead of tedious and time-consuming installations of patched kernels.

Our prototype implementation of ExSched supports Linux and VxWorks
and it comes with example schedulers which include hierarchical and multi-core schedulers in addition to traditional fixed-priority scheduling (FPS) and earliest deadline first (EDF) algorithms.

**Contribution** Shinpei Kato is the main developer of the ExSched core, Mikael Åsberg has developed part of it. Mikael Åsberg’s contribution (in terms of developed schedulers) is limited to the hierarchical scheduler plug-ins (both uni-core and multi-core) and the trace recorder functionality. Shinpei Kato and Mikael Åsberg contributed most to the writing of the paper. Mikael Åsberg was the main driver in writing the paper.

### 5.2 Paper B


**Summary** This paper presents a preliminary study of applying partitioned scheduling in the seL4 microkernel. This microkernel is the first OS kernel ever to be formally proven for its functional correctness. Even though the kernel is completely verified it still delivers high performance comparable to other L4 kernels. The seL4 kernel implements *isolation* of components in terms of the memory resource and security. However, there is still a missing part when it comes to isolation and that is time partitioning. Time partitioning can be implemented inside the kernel (privileged mode) or in user space (user mode). The latter is done using regular user-space thread(s) and can easily be modified while the other approach requires re-verification of the kernel whenever modifications to the time-partitioning policy is done. On the other hand, having the time-partitioning mechanism in privileged mode would yield better performance. We have implemented time partitioning (partitioned scheduling) in the seL4 user space and we elaborate on its performance in terms of overhead costs.

**Contribution** The basic idea was suggested by Mikael Åsberg. Mikael implemented the scheduler, evaluated it in seL4, and wrote the paper.
5.3 Paper C


**Summary**  This paper evaluates the performance of four verified real-time operating-system schedulers. We used the UPPAAL and Frama-C verification tools, and our newly developed code generator TAToC (which is one of the main contributions in this paper), to develop the verified schedulers. These schedulers isolate operating-system threads into time partitions. This technique is referred to as resource reservation and it is a commonly used scheduling technique in the aviation industry since it effectively isolates timing and memory faults within each partition (and simplifies certification). Hence, this will prevent errors from propagating to different parts of the system.

The performance of the synthesised schedulers was evaluated in the seL4 micro-kernel. This kernel is unique since it is the only existing kernel that is 100% verified. The kernel itself uses partitioning as a means to separate critical applications from non-critical ones. The aim of this paper is to investigate and develop a performance-efficient, and verified, scheduler that is suitable for the seL4 kernel, since it currently lacks a resource-reservation based scheduler.

**Contribution**  The idea was suggested by Mikael Åsberg who also did all of the practical work, including the writing of the paper.

5.4 Paper D


**Summary**  Hierarchical scheduling has major benefits when it comes to integrating hard real-time applications. One of those benefits is that it gives a clear runtime separation of applications in the time domain. This in turn gives a protection against timing error propagation in between applications. However, these benefits rely on the assumption that the scheduler itself schedules applications correctly according to the scheduling parameters and the chosen
scheduling policy. A faulty scheduler can affect all applications in a negative way. Hence, being able to guarantee that the scheduler is correct is of great importance. Therefore, in this paper, we study how properties of hierarchical scheduling can be verified. We model a hierarchically scheduled system using task automata, and we conduct verification with model checking using the Times tool. Further, we generate C-code from the model and we execute the hierarchical scheduler in the VxWorks kernel. The CPU and memory overhead of the modelled scheduler is compared against an equivalent manually coded two-level hierarchical scheduler. We show that the worst-case memory consumption is similar and that there is a considerable difference in CPU overhead.

**Contribution** Mikael Åsberg was the main driver in writing the paper, and he was conducting the modelling, verification and synthesis.

### 5.5 Paper E


**Summary** In this paper we present a new synchronization protocol called RRP (Rollback Resource Policy) which is compatible with hierarchically scheduled open systems and specialized for resources that can be aborted and rolled back. We conduct an extensive event-based simulation and compare RRP against all equivalent existing protocols in hierarchical fixed priority preemptive scheduling; SIRAP (Subsystem Integration and Resource Allocation Policy), OPEN-HSRPnP (open systems version of Hierarchical Stack Resource Policy no Payback) and OPEN-HSRPwP (open systems version of Hierarchical Stack Resource Policy with Payback). Our simulation study shows that RRP has better average-case response-times than the state-of-the-art protocol in open systems, i.e., SIRAP, and that it performs better than OPEN-HSRPnP/OPEN-HSRPwP in terms of schedulability of randomly generated systems. The simulations consider both resources that are compatible with rollback as well as resources incompatible with rollback (only abort), such that the resource-rollback overhead can be evaluated. We also measure CPU overhead costs (in VxWorks) related to the rollback mechanism of tasks and
resources. We use the eXtremeDB (embedded real-time) database to measure the resource-rollback overhead.

**Contribution** Mikael Åsberg is the inventor of the RRP protocol. He also did all of the simulations and wrote the paper.

## 5.6 Paper F


**Summary** This paper presents an extensive implementation study where we evaluate and compare different synchronization protocol mechanisms within the domain of two-level hierarchical fixed-priority preemptive scheduling. These protocol mechanisms include HSRPnP (Hierarchical Stack Resource Policy no Payback), HSRPwP (Hierarchical Stack Resource Policy with Payback), SIRAP (Subsystem Integration and Resource Allocation Policy), RRP (Rollback Resource Policy) and SRPwD (Stack Resource Policy with Donation). In an attempt to shed new light to the research in this area, we focus on the actual software implementation of these protocols in a widely used real-time OS (VxWorks). This study is not based on worst-case schedulability analysis which is the most common angle of work in this research field. All five protocols have been implemented, tested and executed for several months with many different parameters, for example; variant number of subsystems, number of resources, system utilization settings, resource allocation strategies etc. These tests generated a large amount of useful data, for example, protocol overhead, effective subsystem utilization, number of protocol mechanism invocations etc. Due to the large complexity and size of this data, we analyzed the data with state-of-the-art statistical methods and tools (Principal Component Analysis) in order to grasp the efficiency of the protocols with respect to a large number of different parameters.

**Contribution** Mikael Åsberg came up with the idea for this paper. He did all of the implementation work, including the writing of the paper.


Bibliography


